Learning Computer Programming using JAVA with 101 examples

Atiwong Suchato

This book is a creation of
the **Knowledge Collection and Contribution Initiatives**
by the Department of Computer Engineering,
Chulalongkorn University

Learning Computer Programming using JAVA with 101 Examples

Atiwong Suchato

LEARNING COMPUTER PROGRAMMING
USING JAVA WITH 101 EXAMPLES
Atiwong Suchato

1. Java (Computer program language).

005.133

ISBN 978-616-551-368-5

First Printing: July, 2011

Published by

Department of Computer Engineering
Faculty of Engineering,
Chulalongkorn University
Phayathai, Bangkok, 10330
THAILAND
http://www.cp.eng.chula.ac.th

This book is a creation of the *Knowledge Collection and Contribution Initiatives* by the Department of Computer Engineering, Chulalongkorn University

To my parents, who have always committed
to providing the best education for me.

Preface

Computer programming skills are currently must-have skills for every university graduate in any fields of Science and Engineering. This book is aimed to be a textbook suitable to be used in a first programming course for university-level students. The primary goals of this book are to introduce students to creating computer programs to solve problems with high-level languages. Programming concepts appearing in modern programming languages are presented through writing Java programs. Java is selected as the language of choice due to its relatively simple grammars. It is also a good choice for introducing students to the concept of object-oriented programming which is one of the most popular paradigms in the current days. Furthermore, Java is one of the most widely-adopted programming languages by the industries.

This book is developed from the class notes that the author wrote for the introductory computer programming course offered to students in the International School of Engineering, Chulalongkorn University. The writing style and the content organization of this book is designed to be straight-forward. Details not crucial to understanding the main materials presented in their related sections are usually omitted in order to relieve the readers from worrying about having to know 'too much'. References for further readings will be given along the way.

The author hopes that this book would introduce readers to the joy of creating computer programs and, with examples given in this book, writing computer programs would appear to be more realizable, especially for beginners with absolutely no programming background.

The source code used in all 101 examples, as well as possible list of errata, can be found on the Facebook page of this book:

http://www.facebook.com/programming101

Typographical Conventions

The following typographical conventions are used in this book:

Italic
: indicates new terms, class names, method names, and arithmetic variables.

`Bold constant width`
: indicates Java keywords, source codes, expressions used in their related source codes.

About the Author

Dr. Atiwong Suchato is currently an assistant professor at the department of Computer Engineering, Faculty of Engineering, Chulalongkorn University. He earned his bachelor degree in Electrical Engineering with the first-class honor from Chulalongkorn University while being ranked in the 1st among the graduated class.

Dr. Suchato received the Anandamahidol foundation scholarship in 1997 to pursue his advanced degrees at Massachusetts Institute of Technology (MIT). In 2004, he received his doctoral degree in Electrical Engineering and Computer Science from MIT and joined the Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University. Since then, Dr. Suchato has been teaching computer programming courses to students in several programs including programs in the International Engineering School (ISE), Chulalongkorn University. He was appointed an assistant dean position overlooking the Information Technology strategies and their implementation in 2008. He was also a key member in the team that initiated the Information and Communication Engineering (ICE) program for the faculty in 2005 as well as the committee revising its curriculum in 2010.

His research interests are in the area of computerized speech and language technologies and their applications to assistive technology.

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1: Introduction to Computer Systems

Objectives

Readers should

- Be familiar with computer components and understand their main functionalities.
- Conceptually understand how data are represented in computers.
- Be able to conceptually explain how to solve problems using computers.

Hello Computer, my dear friend!

A computer is a machine capable of automatically processing data according to instruction lists given. Data processing includes, but not limited to, carrying out arithmetic operations, comparing logical values, as well as, transmitting, receiving and storing information. Data processing tasks, no matter how long or complex they are, can be performed with series of some simple commands, considered "native" to the computers processing those data.

From day one until now, the so-called "native" commands are done electronically in most computers. Many benefits we gain by using computers as our tools to problem solving are due to the fact that electronic devices can perform these commands in a very fast and reliable fashion. Here are some examples of such benefits in normal circumstances.

Computers work fast. Despite common complaints that sounds like "My computer is slow", it does work very fast! A computer can execute billions commands in a second. Suppose we have golf balls that each of them can be either white or black, how fast can you compare the colors of one thousand pairs of golf balls and say how many pairs are different? A computer can do it in one nanosecond.

Computers work consistently. No matter how many times a computer performs the same job, the result is similar every time. (Telling a computer to intentionally does something randomly dose not count!)

Computers remember a huge amount of stuffs. A large amount of data can be written into computers' memory. Hundreds of books can be stored in and recalled back from a computer's memory with ease.

Computers are loyal. Computers do and only do as instructed, straightforwardly. If the instruction is right, the result is right.

Computers work hard. Computers do not complain about doing repetitive tasks. It does not get bored as it happens with people. Surprisingly, repetitive tasks are things that computers do all the time and do well. Furthermore, computers can work continuously for very long periods.

Despite all the good points, ones need to keep in mind that a computer is just another invention of our planet earth. As of the current technological advance, computer does not think by itself and it does not have human's common sense. What we can do with the help of computers are not unlimited, but, trust me; they are more than enough in most cases.

To learn how to "instruct" computers using sets of instructions called "programs" is the main focus of reading this book.

Computers in Our Lives

Although the word "computers" was originally used to refer to a person who performs calculation, nowadays, when coming across the word, ones usually picture machines or devices manufactured in some certain different form factors. Core capabilities of these machines include "automated calculation" and "programmability" with the former infers that the machines can perform some pre-defined calculation procedures automatically and the later infers that there are ways to schedule series of operations that the machines are capable of. In computers' early days, the two capabilities of the machines were enabled using mechanical

components while modern computers mainly rely on combinations of electronic devices and mechanical parts. Advancements in many technological fronts contribute to improved computer capabilities. The "brain" of a computer was used to be a room filled with vacuum tubes and large mechanical parts. It is now a package of wired electronic devices with its size smaller than a matchbox due to advancements in Semiconductor technologies. Paper-based punched cards were once used to stored data which are represented by series of holes punched through paper cards. Today's data storages still more or less adopt such a concept of data representation but with significantly reductions in time and space requirements. For example, magnetic characteristics of a miniscule region on the surface of a hard disk are manipulated instead of punching a hole on a paper card. This allows data processing to be performed more quickly and efficiently.

Figure 1 is a picture of the ENIAC, one of the first computers invented. Its programmability is achieved by physically re-wiring its circuits. Luckily for us who are now in the 21st century, computers are far more powerful and, although their enabling technologies are complex, they are far less complicate to use.

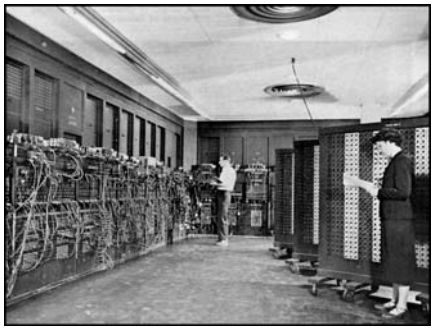


Figure 1: ENIAC
(This picture is a work of U.S. Army Photo and is available in the public domain.)

Today's computers come in many different appearances, sizes, and capabilities. Some are general-purposed computers, while some are designed or pre-programmed for more specific tasks. Desktop and notebook computers together with popularity-gaining "slate" or tablet

computers are probably among the most recognized forms of computers in our everyday lives. However, do not ignore the fact that specific-purposed computers embedded in appliances are far from rare. We sometimes just do not notice one.

Desktop Computer
(picture from www.dell.com)

Notebook computer
(picture from www.sonystyle.com)

Tablet computer
(picture from www.hp.no)

Portable game console
(picture from www.nintendo.com)

Smart mobile phone
(picture from www.apple.com)

Washing machine with embedded computer
(picture from www.lg.com)

Figure 2: Different types of computers

What Computers do

Every form of computers has the same basic structures of operation. First, they must have ways of taking input data into the systems. Data

supplied into computer systems could be in various forms such as keystrokes, text messages, images, actions and etc. No matter which forms they take, they will be converted into a uniform representation, which will be discussed later in this chapter, and stored in the memory of the systems. Then, computers process the data by reading from the memory and placing the processed results back. Finally, they present outputs to users or components outside their systems. During the process, if computers need to store some data more permanently, they can write those data into storages which are either integrated in the systems or located externally outside their systems.

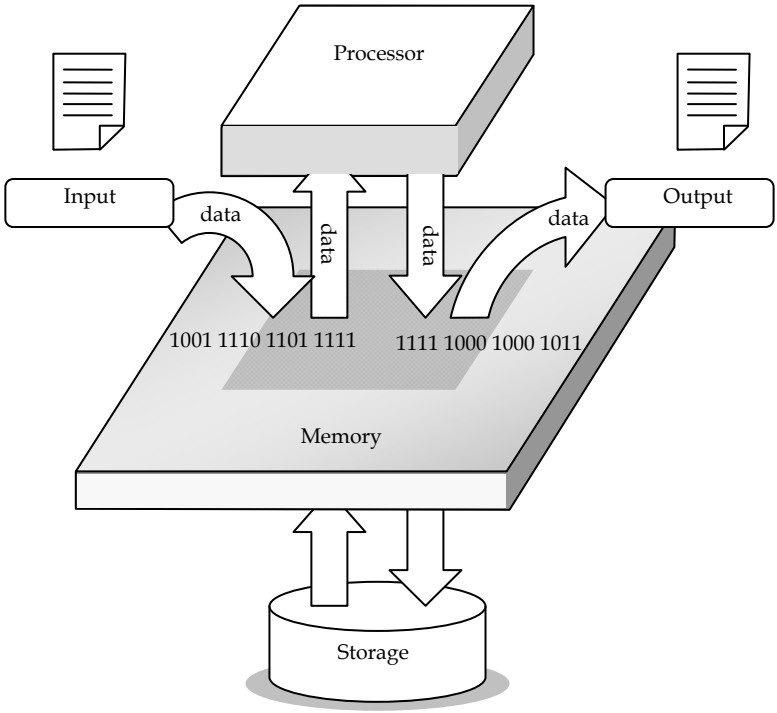


Figure 3: Conceptual diagram of how a computer works

Do computers need memory in order to process data? The answer is "Yes, they do." Memory is used for storing data either temporarily or permanently in order to facilitate the desired data processing task. Let's

take performing an Arithmetic calculation as our example of data processing. Suppose that we are to compute the result of multiplying two numbers by ourselves, manually. Before our brains can perform such multiplication, somehow, we have to memorize the two numbers first. If the numbers are too large or the multiplication is too complex, we might have to jot them down on a piece of paper. Memorizing or writing down those numbers is analogous to the computer placing data into its memory waiting for the data to be processed. Furthermore, the steps or sequences of actions required for executing the calculation must also be registered in the memory so that they can be retrieved in order for the computer to figure out what to do for achieving the calculation result, which, once obtained, will be written into the memory too.

Electronic devices and mechanical parts of computers are called *hardware*, while contents that instruct computers what to do are called *software*. Information or messages that do not directly control computer hardware are considered *data*.

Hardware

Computer hardware can be grouped into three groups according to their functionalities. They are:

- Central Processing Unit (CPU)
- Memory
- Input/Output (I/O) device

Given a program, every components work together according to the instructions in that program.

Central Processing Unit

Central Processing Unit (CPU) is responsible for execute given instructions. It is the brain that does arithmetic and logic processing, as well as control other hardware coordination. A CPU is typically a small chip whose functionalities are done via the controlling of electric current flowing through the miniscule integrated circuit inside the chip.

Different generations and brands of CPUs differ in their architectures and speed of operation. We often refer to different CPUs by their commercial names, such as Intel® Pentium® M, Intel® Core™ 2 Duo, Intel® Core™ i7, Intel® Atom™, Intel® Itanium®, AMD Athlon™ 64, AMD Opteron™, AMD Phenom™, Sun UltraSPARC, PowerPC, and etc.

Each of most CPUs requires a clock signal that synchronizes the CPU's processes as well as other hardware devices. Generally, the faster the clock signal, the more steps in the processes to be executed in certain time intervals. Clock speeds are measured in the unit *hertz* (Hz). The higher the clock speed, the more steps a CPU can perform in one second.

Memory

Memory is responsible for storing data and program. There are two types of memory grouped by the purposed task for which that type of memory is used. The first type is called *main memory*, which is a temporary storage used for storing instructions and data currently in use. This type of memory is usually more expensive but faster than the other type of memory, called *secondary storage*. This type of storage is used for storing data that is wished to be remembered permanently, or at least, longer than the one stored in the main memory. Typically, such data might be user's data, for example document files, multimedia files, and archived data etc. Secondary storages are slower but cheaper than main memory.

Figure 4: A 2GB DDR2 RAM with integrated heat sinks.
(The picture was publicly distributed at http://commons.wikimedia.org by Victorrocha, retrieved on Sept 13th, 2010)

Floppy disk

Compact Disc

Flash memory device

Figure 5: Different types of secondary storages

Examples of main memory are various types of Random Access Memory (RAM), while Floppy disks, Hard Disk Drives, Compact Disc (CD) such as CD-ROM, CD-R, CD-RW, etc., Digital Video Disc (DVD) such as DVD-ROM, DVD-R, DVD+R, DVD-RW, etc., and flash memory devices belong to the secondary storage.

I/O Devices

I/O is abbreviated for Input/Output. Input devices are responsible for taking data or information into computer systems. In contrary, output devices are responsible for providing data or information to the outside world. I/O devices can be used between computers and their users, as well as among computer systems. Devices that can transmit data among computer systems are also considered I/O devices.

Input into and output from computers can be of many forms. More traditional forms of input information could be keystrokes made via computer keyboards and movements of computer mice, while recent trends in novel input forms come in the forms of multi-finger gestures made on some input surfaces, live video input via web cameras, velocity and acceleration measured by accelerometer embedded in input devices of various forms, as well as voice commands spoken by human users. On the output side of the story, nowadays output information are presented to the users by many more means beyond graphical displays of computer monitors or printed media made by printers. Information encoded in rhythms of mechanical vibrations made on a device hold by the users serves as an example belonging to this case. Considering a

device from its whole package, we usually can find devices that provide both the input and the output.

Figure 6: The Wiimote, a device capable of delivering many forms of input and output such as button pressing, position and acceleration of the device, lamp signals, vibration as well as audio output
(The picture was publicly distributed at http://commons.wikimedia.org by Asmodia, retrieved on September 15th, 2010. This file is licensed under the Creative Commons license. (free to share and/or to remix))

Software

Software is a set of instructions that is used for controlling computer hardware. Some software are designed to communicate directly with computer hardware, while some other software are not, but they operate on top of other software, such as operating systems (OS), that acts as their delegates in communicating with hardware. Conceptually, the relationship among users, application software, system software, and hardware can be shown as in Figure 7. The users interact with computers by having physical contacts with the hardware parts as well as consume the output presented via the hardware. Application software normally focuses on supporting users to accomplish the tasks they desire, while system software takes care of providing environments of the computer systems that are required for other applications to run as expected.

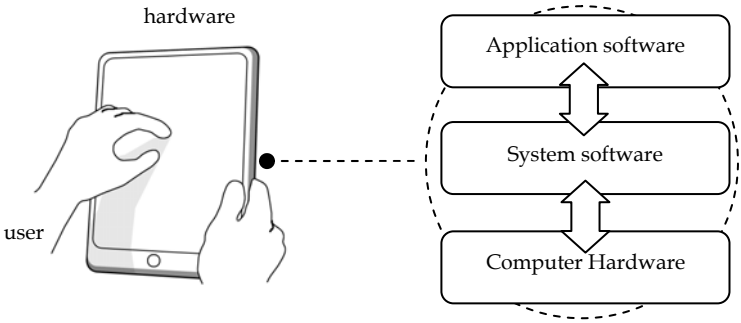
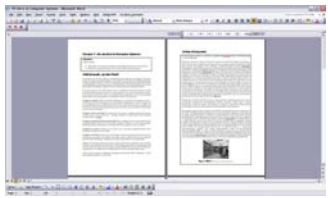


Figure 7: Relationships among users, application software, and system software

Application Software Vs. System Software

Application software covers programs that help user in accomplishing specific tasks such as word processors, web authoring tools, computer games, and etc. System software usually covers programs that are not directly intended to help accomplishing the user's task but supports other application software. System software could involve providing needed functionality for application software, maintaining computer systems so that they are in suitable condition, facilitating application software with productivity-related services, or even building other application software. Examples of system software are operating systems, system tools (e.g. antivirus, archiving tools, disk defragmenter), and software development tools (e.g. compilers, debuggers, integrated development environment suites).

Photo editing software



Word processing software

Figure 8: Screenshots of some application software

Operating System (OS)

An *operating system* (OS) is a system software that operates resources, both hardware and software, of a computer. An OS usually pre-define basic functionalities that are common to most applications and provide such functionalities, such as file and device management, graphical user interface rendering, and computer networking, to applications running on that OS. This makes the development of application software easier and faster. Examples of well-known OS's include various versions of Microsoft Windows, MacOS, and other Unix-based OS's. In current computing trends, OS's designed for portable electronic devices are gaining more influences. Some examples of these OS's are variations of the Andriod OS as well as the iphone OS.

Binary Representation of Data

In today's computer systems, places where data as well as programs can be stored are manufactured in various ways. However, what is common to most of them is the fact that, logically, they are designed to hold a long sequence of "blocks" where each of the blocks, when ready, can be in either one of two different states. We call each one of these blocks as a *bit* and we usually refer to the two different states as the state '0' and the state '1'. In other words, we say that each bit takes a binary value which can be either 0 or 1. Different combinations of the binary values associated with a bit sequence are used to represent different values stored in the location corresponding to those bits. The amount of eight bits is called one *byte*. The capacity of computer storage is measured in numbers of bits or numbers of bytes.

Figure 9: An example binary representation of 1-byte data

In old days, these blocks were implemented using mechanical relays. Today, they were implemented based on smaller (by several

magnitudes) devices utilizing electrical, optical, and other properties of fabricated materials.

Different types of data, including but not limited to numerical data, alphabetical data, and logical data, are represented in forms of sequences of binary values using different encoding schemes. With a given encoding scheme, a value can be converted correspondingly to its binary representation and written into the memory. Unsurprisingly, with the encoding scheme known, the binary values of a bit sequence can be read from the memory and converted back to its original form.

For positive integer values, each value is stored in the memory using its corresponding representation in base 2, E.g. the integer value 'eleven (11)' in the decimal system (base 10) is stored in the memory as 1011_2. We usually neglect the subscript 2 when it is obvious.

It is not within the scope of this book to cover encoding schemes for various data types being used in computer systems.

The Power of Two

Since computers represent all data and instructions in binary formats, we usually run into cases when the value of 2^n, where n is a positive integer, is needed. For example, we might want to calculate how many different values can be stored in n bits of memory. Being familiar with the values of 2^n where n is a small positive integer can help us come up with answers to some questions arising when we design our programs more quickly. Adept programmers, computer scientists, and other computer-related professionals use these numbers so often that they can naturally tell their values without having to pause and calculate.

The following table lists the values of 2^n from n being 0 to n being 11. It is not much of a burden to memorize these values.

n	Values of 2^n	n	Values of 2^n
0	1	6	64
1	2	7	128
2	4	8	256
3	8	9	512
4	16	10	1024
5	32	11	2048

Table 1: Values of 2^n

Units of Measure

When measuring capacity of computer storage, people add prefixes such as *kilo*, *mega*, and *giga* in front of the byte unit to avoid explicitly using large number. The commonly used prefixes, *SI prefixes*, are based on powers of ten. For SI prefixes, kilo means one thousand (10^3), mega means one million (10^6), giga means one billion (10^9), and tera means one million millions (10^{12}). Thus, the distance of 1 kilometer means 1,000 meters. However, in the computer industry, another type of prefixes which are *Binary prefixes* are also used. In this latter system things are counted using powers of two. The binary prefixes become as shown below. E.g.: The capacity 1 Gigabyte (GB) equals to 1,073,741,824 bytes.

Binary Prefix	Power of two	Value
kilo (k)	2^{10}	1,024
mega (M)	2^{20}	1,048,576
giga (G)	2^{30}	1,073,741,824
tera (T)	2^{40}	1,099,511,627,776

Table 2: Binary prefixes

Example 1: How large is a giga byte?

Given that binary prefixes are used, how many kilobytes are there in 1 gigabyte?

There are 2^{30} bytes in 1 Gigabyte and 1 kilobyte is 2^{10} byte. Therefore, there are $2^{30}/2^{10} = 2^{(30-10)} = 2^{20} = 1{,}048{,}576$ kilobytes in 1 gigabyte.

Example 2: Memory card capacity

Suppose that a digital camera invariably uses a space of 256kB to store an image and it also reserves a space of 20kB per each memory card to store some camera specific data. How many images can be stored on a blank 512MB memory card?

The remaining space of the card once the space required for the camera specific information is deducted in kB is (512×2^{10})-20 kB. Let n be the number of images. Therefore, the maximum value of n is the largest integer not exceeding

$$\frac{(512 \times 2^{10}) - 20}{256} = 2 \times 2^{10} - \frac{20}{256} = 2^{11} - \frac{20}{256}.$$

Recall that 2^{11} is 2,048, and we know that 20/256 is a positive value less than 1. That makes 2,048 – 20/256 being 2,047 plus some fraction of 1. Therefore, the maximum value of integer n in this case is 2047, or this memory card can hold 2,047 images when using with this digital camera.

Problem Solving Using Computer Program

When you, as a novice programmer, would like to solve some problems by writing a computer program to solve them, you should have a set of steps about how to do it in your mind. Different people may take different steps. Experienced programmers are likely to deliver programs (which are not too big) out of pure intuitions. However, for those with less experience, it does not hurt to follow some steps. Here, we introduce the following steps when developing a computer program for some problems.

Problem defining: Make sure you understand the thing you want to solve. You want to make sure that your efforts are spent in solving the real problem. A clever solution for the wrong problem is a waste!

Analysis: Determine inputs, outputs, and relevant factors for the defined problem.

Design: Describe the methods used for solving the defined problem. Provide a detailed process required to be done in order to go from taking the inputs to producing the outputs.

Implementation: Write the code corresponding to the design. You can always go back to improve earlier steps so that the code can be written in an efficient fashion.

Testing: Test the program whether it delivers the correct results. You may need to prepare some test cases, in which you know the expected outputs for a set of test inputs. If any outputs do not appear as expected, go back to the analysis and design steps. Once you are sure that they are done correctly, the problem is likely to be caused by 'bugs' in the code. Never deliver your code without carefully testing it.

Let's look at an example scenario that we will use it to walk through these steps in creating a computer program that solves an electrical circuit problem.

Example 3: An electrical resistive circuit problem

Suppose that we would like a way to quickly find the amount of current flowing through the circuits shown below in Figure 10, in which the voltage source is fixed at 220 Volts while the amount of the resistance in the circuit could be set to any positive value in Ohms.

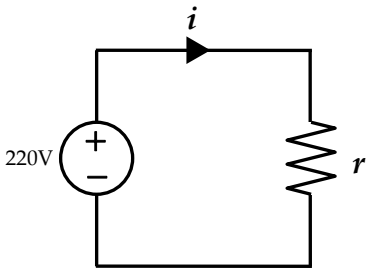


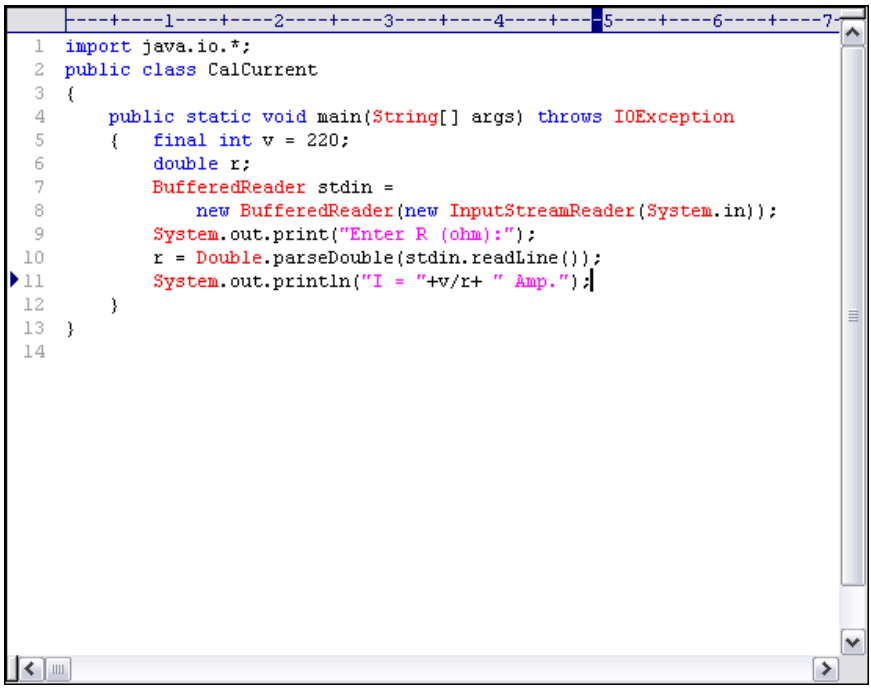
Figure 10: An electrical circuit with a resistor connected in series with a voltage source of 220 Volts.

Problem defining: A program that calculates the amount of current in the circuit with a voltage source of 220 Volts is connected in series with a resistor whose resistance value will be defined at run-time, i.e. when the program runs each time.

Analysis: Since the value of the resistance needs to be defined each time the program runs, that value should be the input to the program. Let's decide that the input must be typed in correctly by the user of the program via keyboards once prompted by the program. From the problem definition, the output will be the corresponding amount of current flows in the circuit. We will have the program show it on screen. Another relevant factor is the amount of the voltage supplied by the voltage source. This is always 220 Volts.

Design: From what we have learned from Circuit theory (or at least from what I am about to tell you now), to obtain the output which is the amount of current in Ampere, i, flowing through a resistance of r Ohms connected in series with a 220V voltage source, we rely on the equation: i = 220/r. Therefore, once the input value r is obtained, the program must calculate the value of 220/r and show the result on screen.

Implementation: The source code of a Java program that is capable of doing what are described above looks like the one in Figure 11. How to come up with such a program (as well as more complicated Java programs) is the main focus of the rest of this book.



```
1  import java.io.*;
2  public class CalCurrent
3  {
4      public static void main(String[] args) throws IOException
5      {   final int v = 220;
6          double r;
7          BufferedReader stdin =
8              new BufferedReader(new InputStreamReader(System.in));
9          System.out.print("Enter R (ohm):");
10         r = Double.parseDouble(stdin.readLine());
11         System.out.println("I = "+v/r+ " Amp.");
12     }
13 }
14
```

Figure 11: A screenshot showing the source code of a Java program

Testing: Once the program is written. We have to compile and run the program with a reasonable amount of test cases to ensure that the program does not contain any *bugs* which are error caused by the program. If there are bugs in the program, we re-consider our design and implementation and correct them. This process of correcting error is called *debugging*.

Methodologies used in precisely testing a program are beyond the scope of this book. Interested readers should learn further in additional readings such as [Pre2010] and [Mcc2009]

In this book, we will follow the problem solving steps presented here in some examples that the steps are not trivial. In others, we will mainly

focus on the details of the programs which are the deliverables of the implementation step. Discussion will be made based on every source code and relevant information presented in all examples.

Exercise

1. For each of the following fields, give an example of how computer can be used to assist human's job. (Be creative!)

- Automobile manufacturing
- Education
- Healthcare
- Homeland security
- News agency
- Retail business

2. Which part(s) of a computer that you think its/their functionalities are the closest to human's brains?

3. How is the speed of a CPU measured?

4. Name two devices that can act as both input and output devices.

5. How many bits are required to store the decimal value 5,000?

6. A computer has 512MB RAM. How many bits can be stored at once in its RAM?

7. How many Megabytes are there in a Gigabyte?

8. Argue whether each of the following software is application software or system software.

- Antivirus
- Games
- Calculator
- Instant messenger
- Java runtime environment
- Operating system

9. Describe in brief what the following applications do.

- Web browser
- Word processor

- Media player
- Instant messaging program

10. Analyze and design a computer program that find the average of 5 numbers entered by the user of the program.
11. Suppose you are to write a currency converter program that can convert the amount money among Baht, US Dollar, and Euro. Describe how you would analyze the problem and design the program to be written.

Chapter 2: Programming Concepts

Objectives

Readers should

- Know the steps required to create programs using a programming language and related terminology.
- Be familiar with the basic structure of a Java program.
- Be able to modify simple Java program to obtain desired results.
- Be able to use flowcharts to describe program processes.

Programming Languages

A *program* is a set of instructions that tell a computer what to do.

We *execute* a program to carry out the instruction listed by that program.

Instructions are described using *programming languages*.

High-level programming languages are programming languages that are rather natural for people to write. Examples of high-level programming languages include Java, C, C++, C#, Objective-C, Visual Basic, Pascal, Delphi, FORTRAN, and COBOL. Still, these names are just a small fraction of high-level programming languages available nowadays for the general public to use.

The most primitive type of programming language is a *machine language* or *object code*. Object code is a set of binary code that is unique to the type of CPU. Each instruction of the object code corresponds to a fundamental operation of the CPU. That means it usually requires many object code instructions to do what can be done in one line of high-level language code. Writing object code directly is tedious and error-prone.

High-level language code does not get executed on a computer directly. It needs to be translated from the high-level language to suitable machine language first. Such a translator program is normally referred to as a *compiler*.

The input of a compiler is referred to as *source code*. Source code is *compiled* by a compiler to obtain *target code*.

Running a Java Program

Unlike other programming languages, compiling Java source code does not result in a machine language program. Instead, when Java source code is compiled, we get what is called Java *bytecode*. Java bytecode is a form of machine language instructions. However, it is not primitive to the CPU. Java bytecode runs on a program that mimics itself as a real machine. This program is called the *Java Virtual Machine* (JVM) or *Java Run-time Environment* (JRE).

This architecture makes Java bytecode runs on any machines that have JVM, independent of the OSs and CPUs. This means the effort in writing Java source code for a certain program is spent once and the target program can run on any platforms. (E.g. Windows, MacOS, Unix, etc.)

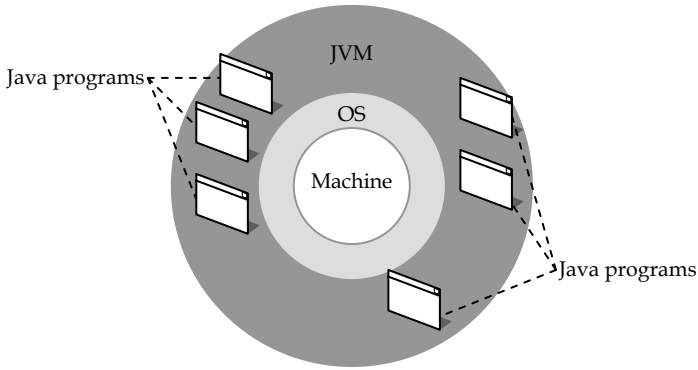


Figure 12: Platform-independent architecture of Java language

Typical Programming Cycle

During the implementation of a program, programmers always run into a typical cycle, shown in Figure 13.

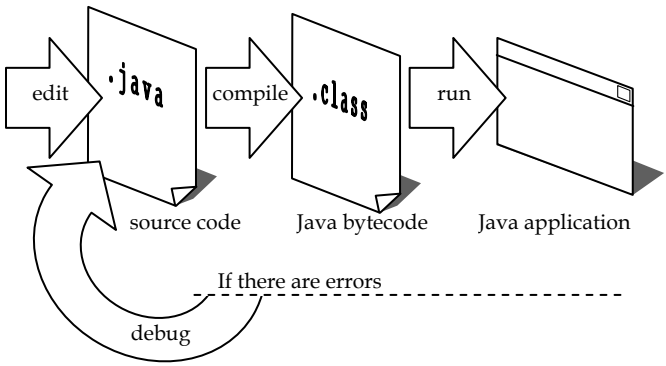


Figure 13: Typical cycle in Java programming

It normally starts with the coding step, in which the source code is written in any text editors or integrated programming environment. In Java, source code files are usually saved with *.java* extension. Once the Java source code is saved, it is compiled using a java compiler, such as *javac.exe*, to obtain the resulting Java bytecode, which is also called a Java *class*. The actual Java class file is created with *.class* extension. Then, the program is executed by running the command '*java.exe*' on the class file. If the result appears as expected, the cycle terminates. Otherwise, the source code has to be edited, then compiled and executed again. The process of fixing the source code in order to obtain the right result is called 'debugging'.

Preparing Java Computing Environment

Getting the required software

A necessary step before ones can write computer programs to help solve some problems is to prepare the required computing environment. Once a programmer decides that he or she will use a specific programming language (or an application development framework) to create computer programs, certain sets of software needs to be installed. For a high-level programming language, these usually include a text editor, which lets programmers type in the instructions of the programs and save into a text file, and a software that can change that text file into a file filled with binary representations that can be understood by the operating systems of the computer, upon which the programs are aimed to be run. Since the platform-independent nature of Java relies on the fact that the Java Virtual Machine (JVM) must be there to nullify the differences among different operating systems, there is another piece of software that must be installed on the machine apart from the two already mentioned.

Any text editors in the market would work for writing Java programs. For beginners learning Java as their first programming language, a simple text editor would suffice. In fact, beginners are encouraged to use a really simple text editor such as notepad for their first few programs rather than using a sophisticated Java-enabled editor that automatically creates some mundane parts of Java code. Getting your hands dirty by actually typing in every characters required in the Java code should help you learn Java programming more profoundly, especially once different parts of the Java code are demystified later on in this book.

Apart from the text editor which ones can choose according to their preferences, probably, the easiest way to obtain the rest of the required software is to download and install the Java Development Kit (JDK) from the official Java website. Due to the quickly changing nature of computer business, the website's specific URL might be changed over years. The trick in finding it is to simply perform a search for "JDK download" on a search engine.

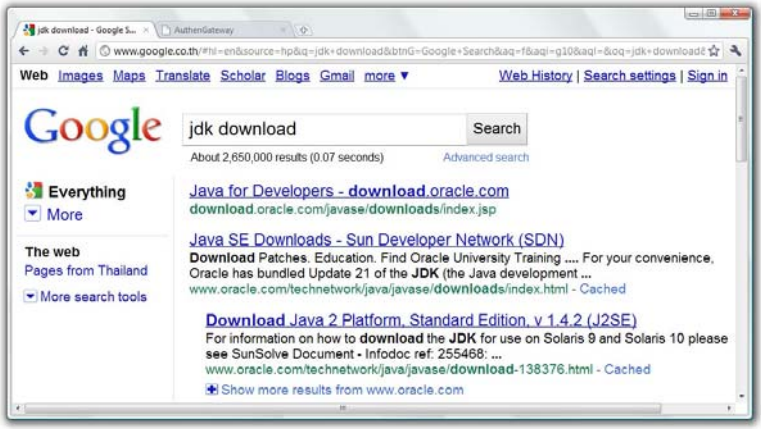


Figure 14: Sample search results for the keywords "jdk download"

The Java platform has many editions offered to be downloaded and installed such as Java SE, Java Embedded, Java EE, Java ME, and etc. For general purpose Java programs, Java SE (Java Platform, Standard Edition) would be the most suitable one.

The JDK package contains many things useful for developing a Java program as well as the Java Runtime Environment (JRE) that is needed for running any Java programs developed by anyone. When JDK is installed on a machine, it also installs JRE too. Note that for ordinary users that may wish to just run a Java program, they only need to install JRE.

Letting Your OS Know Where to Find Java

One way to compile the source code of a Java program and to run the compiled Java bytecode on JVM is to execute certain programs from the command prompt of the computer's operating system. The Java compiler is the program file with the name *javac.exe*, and the Java interpreter, which is the program to be called when executing any compiled Java bytecode, is the program file with the name *java.exe*. The former one can be founded in a subdirectory called *bin* (short for binaries) of the folder

that JDK is installed, while the latter one is in the bin subdirectory of the folder that JRE is installed.

How to open the command prompt differs among different operating systems. In MS Windows, the command prompt can be found from the start menu, while in MacOS as well as other Linux-based operating systems, the command prompt are basically the *Terminal* program.

For the purpose of learning and practicing writing Java programs, we usually save our source codes and run the corresponding programs in directories different from the ones that JDK and JRE get installed. We generally want to call *javac.exe* and *java.exe* directly from the directories that the source codes are saved, or in other words, the directories in which we are currently working and avoid providing the full path to both *javac.exe* and *java.exe* each and every time we need to compile and run Java programs. Therefore, in order to do that, we need to let the operating system know where to look for programs that are called from the command prompt and cannot be found in the current working directory.

Readers should consult manuals of their operating systems of how to set the paths that they should look inside for unfound programs. For MS Windows, these paths can be registered by setting the environment variable called *path* to include the paths to the *bin* subdirectories of both the JDK and the JRE folders.

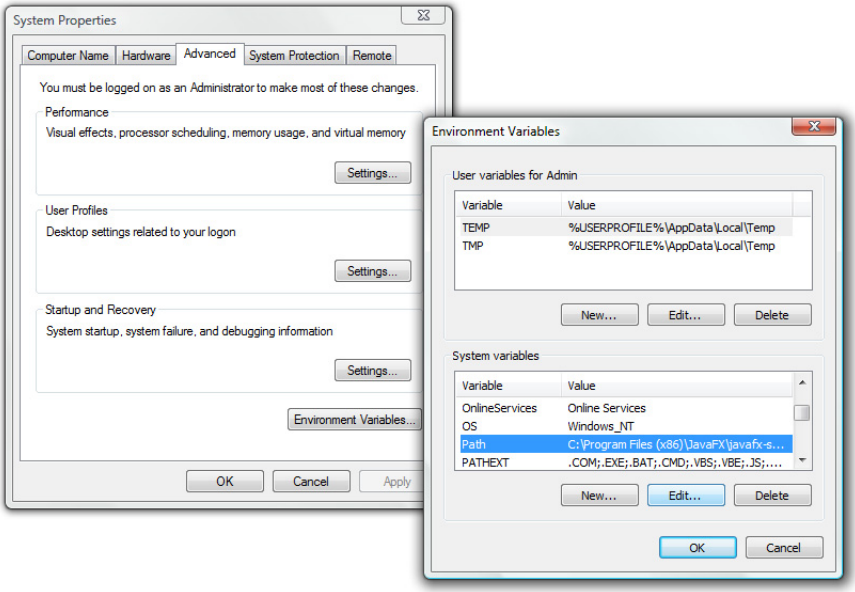


Figure 15: Windows in an MS Windows operating system for setting up the system's environment variables, including the system variable called "Path" which is highlighted in the right window

Compiling and Running Java Programs

To compile a Java source code, the *javac.exe* program could be called from the command prompt. The syntax used here is:

```
javac [Sample.java]
```

where **`[Sample.java]`** should be replaced with the filename of the Java source code to be compiled. Programmers should always be aware of the cases (uppercases or lowercases) of each letters used in the filename and be precise about them. Although in some operating systems, including MS Windows, the cases of every character are ignored. Uppercase and lowercase characters are considered different in some systems. If the latter is the case, it would make the file whose name is Sample.java

different from the file named sample.java and also different from Sample.Java.

To run or execute the compiled Java program, we call the *java.exe* program using:

```
java [Sample]
```

where **[Sample]** should be replaced with the program name which is identical to the name of your source code without its extension (without .java). Note that, at the execution time, the file that is actually needed is the compiled bytecode which is named with the name of the original source code but with .class extension. Figure 16 shows a screenshot of a command prompt in an MS Windows operating system when a Java program called is compiled by *javac.exe* and run by *java.exe* respectively. Note that the sign ">" appearing on screen is part of the command prompt.

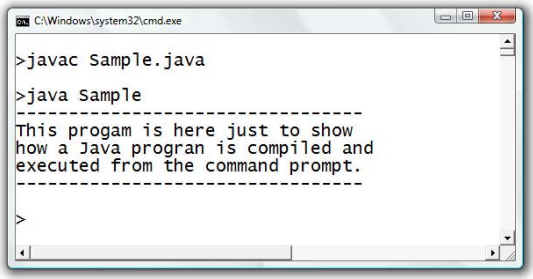


Figure 16: Compiling and running a Java program from the command prompt

Integrated Development Environment (IDE)

An Integrated Development Environment (IDE) is a program that facilitates steps in computer program development. An IDE usually provides all functionalities a programmer would need in developing a computer program. It usually consists of:

- a built-in source code editor that highlights and colors texts according to the keywords and syntactic constraints of chosen programming languages,

- compilers and related tools of chosen programming langauges as well as user interfaces for using them, including but not limited to a console showing output and related messages,

- and a debugging tools facilitating users in debugging programs with functionalities such as stepping through each instruction and variable monitoring.

There are many IDEs that support Java and can be downloaded and used for free. Some among the most popular ones are Eclipse (http://www.eclipse.org) and NetBeans IDE (http://www.netbeans.org). Some guides to using these IDEs can be found in [Bur2005] and [Dan2004] for Eclipse and [Mya2008] for NetBeans.

Basic Program Structure in Java

Let's look at a source code named MyFirstProgram.java below. Note that the number at the end of each line indicates line number. They are not part of the source code.

```
public class MyFirstProgram                                    1
{                                                              2
       //This program just shows message on a console.         3
       public static void main(String [] args)                 4
       {                                                       5
           System.out.print("Hello World!");                   6
       }                                                       7
}                                                              8
```

When compiled, this program shows the message "Hello World!" on screen. Now we will investigate the structure of this Java source code in order to get familiar with the basic structure.

First of all, we can observe that there are two pairs of curly brackets {}. The opening curly bracket on line 2 and the closing curly bracket on the

last line make a pair. The ones on line 5 and line 7 make another pair locating inside the first one. Here, we will refer to the content between a pair of curly brackets together with its associated header, i.e. **`public class MyFirstProgram`** for the outer pair or **`public static void main(String [] args)`** for the inner pair, as a *block*. The idea of looking at the program structure as blocks can be illustrated in Figure 17.

Figure 17: Blocks visualization in MyFirstProgram.java

Block 1 is the definition of the program. Block 2 is a part of the definition which is called the *main()* method. The symbol // is used for indicating that any texts after that until the end of the line are comments, which are ignored by the compiler. They are just there for programmers to make notes for themselves or other people reading their source code. Thus, we can say that the definition of this program consists of a comment on line 3 and the *main()* method from line 4 to 7.

The *main()* method is the program's entry point. A program is executed following the commands listed in its *main()*, starting from the top. From this example, when the program is executed, it is the statement on line 6 that prints the message "Hello World!" (without double quotes) on screen.

In Java, what we do all the time is creating what are called *classes*. In this MyFirstProgram.java source code, we create a class named MyFirstProgram. Each class that is created in Java can have one or more *methods*, which are units of code that perform some action. They can be generally thought of as subprograms which appear in the forms of *functions*, *procedures*, or *subroutines* in other programming languages. A

class whose definition contains the *main()* method can be executed, or, in other words, is a program. Classes that do not have the *main()* method cannot be executed. They are there for other program to use.

Now that you have heard about the term *class*, Block 1 in the above example is called the class definition of the class named *MyFirstProgram*, which is a Java program since it has the *main()* method. The statement **`public class MyFirstProgram`** indicates the starting of the class definition, while the statement **`public static void main(String [] args)`**explains how the method named *main()* should be used. We will go into details about these later.

Note that Java requires the source code to be saved using the same name as the class contained in the file (with the extension .java).

Below is an example of a Java program that is perfectly correct but seems to do nothing at all.

```
public class MyFirstProgram                                    1
{                                                              2
        public static void main(String [] args)               3
        {                                                      4
        }                                                      5
}                                                              6
```

Syntax, Keywords, and Identifiers

Writing program in Java, or other computer programming languages, is like writing in any human spoken language but a lot more formal. When constructing a sentence in a human language, we need to follow its grammar. Programming languages also have syntax or rules of the language and they have to be followed strictly.

Keywords are words that are reserved for particular purposes. They have special meanings and cannot be changed or used for other purposes. For example, from the class definition header in the last example, **`public class MyFirstProgram`** , the word **`public`** and **`class`** are keywords. Thus,

both words cannot be used in Java programs for naming classes, methods, or variables.

Identifiers are names that programmers give to classes, methods, and variables. There are certain rules and styles for Java naming, which we will discussed in details later in this book. For the previous example, **`public class MyFirstProgram`**, the word *MyFirstProgram* is a Java identifier selected as the name of the class.

Note that Java is a *case-sensitive* language. Uppercase and lowercase letters are not considered the same. Java keywords are only consisted of lowercase letters.

Comments

It is a good practice to always add meaningful comments into the source code. There are two ways of commenting in Java source code. The first way is called *single line commenting*. It is done by using double slashes // as shown previously. Any texts between and including // and the end of the line are ignored by the compiler.

The second way is called *block commenting*. In contrary to the first way, this can make comment span more than one line. Any texts that are enclosed by a pair of the symbol /* and */ are ignored by the compiler, no matter how many lines there are.

The following program, MyCommentedProgram.java, yields the same result as MyFirstProgram.java. Lighter texts are the parts commented out.

```
// An example of how to add comments                          1
                                                              2
public class MyCommentedProgram                               3
{                                                             4
        //Program starting point                              5
        public static void main(String [] args)               6
        {                                                     7
            System.out.print("Hello World!");                 8
            //This part of code is commented out.             9
            /*                                                10
            System.out.print("Hello again.");                 11
            System.out.println();                             12
            */                                                13
        }//end of main()                                      14
}// end of class                                             15
```

Be Neat and Organized

Whitespace characters between various elements of the program are ignored during the compilation. Together with comments, good programmers use space, new line, and indentation to make their program more readable.

Observe the differences between the following two source code listings.

```
public class ExampleClass{
public static void main(String [] args){
           double x=5;double z,y=2;z=x+y;
System.out.println(z);
}}
```

```
public class ExampleClass
{
       public static void main(String [] args)
       {
           double x=5;
           double z,y=2;
           z=x+y;
           System.out.println(z);
       }
}
```

Both perform exactly the same task. Still, it is obvious that the bottom one is easier to read and follow. Indentation is used in the bottom one, so that the lines in the same block have the same indentation. And, the indentation increases in deeper blocks. Curly brackets are aligned so that the pairing is clear. Also, the programmer tries to avoid multiple operations on the same line.

A First Look at Methods

Just a reminder, we mentioned that methods are units of code that perform some actions. Here, without going too deep into the details, we will take a look at two frequently-used methods. They are the method *print()* and *println()*.

You might notice that when we talk about methods, there is always a pair of parentheses () involved. It is not wrong to say that when you see an identifier with a pair of trailing parentheses, you can tell right away that it is a method name. When invoking a method, we put arguments that are required for the action of that method inside its associated pair of parentheses. What to put in there and in which order are defined when the method is made. Right now, we will not pay attention to the making of new methods just yet.

The method *print()* and *println()* are methods that print some messages onto the screen. They are defined in a standard Java class called *System*. When we say standard, it means that we can just use them and the compiler will know where to look for definitions of those standard methods. The *print()* and *println()* method are invoked by using:

```
System.out.print(<message you want to print>);
System.out.println(<message you want to print>);
```

If you want to show a string of character on screen, replace **<message you want to print>** with that string of characters enclosed in double quotes. If you replace **<message you want to print>** with a numeric value without double quotes, both methods will show the number associated with that numerical value. The different between the two methods is that method *println()* makes the screen cursor enter the next line before executing other instructions.

Example 4: Trivial printing samples

```
public class PrintDemo
{
       public static void main(String [] args)
       {
           System.out.println("");
           System.out.println("          X");
           System.out.println("         * *");
           System.out.println("        *   *");
           System.out.println("       * o   *");
           System.out.println("      *     v *");
           System.out.println("     *  v      *");
           System.out.println("    *        o  *");
           System.out.println("     ***********");
           System.out.println("      ___|_|___");
       }
}
```

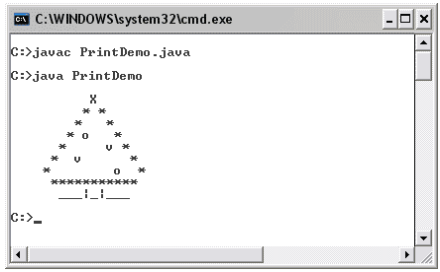


Figure 18: A Java program printing a Christmas tree

The *PrintDemo* program simply shows a simple Java program that invokes *println()* many times in order to print some patterns on screen line by line.

Here is another example program.

```
public class PrintDemo2                                          1
{                                                                2
       public static void main(String [] args)                  3
       {                                                         4
           System.out.print("          ");                       5
           System.out.println(299);                              6
           System.out.println("+         800");                  7
           System.out.println("-------------");                   8
           System.out.print("         ");                        9
           System.out.println(299+800);                          10
           System.out.println("==============");                  11
       }                                                         12
}                                                                13
```

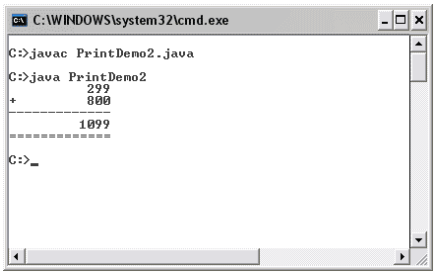


Figure 19: Demonstration of using print() and println() with both String and numeric input

In the *PrintDemo2* program, you should notice that different types of value are used as inputs for the *print()* and *println()* methods. A sequence of characters enclosed by double quotes is called a *string*. Strings (which will be discussed later in this book that they are objects of a class called *String*) are used as inputs on lines 5, 7, 8, 9, and 11. On lines 6 and 10, numeric values are used as inputs to the println() method.

Escape Sequences

An escape sequence is a special character sequence that represents another character. Each of these special character sequences starts with a backslash, which is followed by another character. Escape sequences are used to represent characters that cannot be used directly in a string (double quoted message). Some escape sequences are listed in the table below.

Escape sequence	Represented character	Escape sequence	Represented character
\b	Backspace	\\	Backslash
\n	Newline	\"	Double quote
\t	Tab	\'	Single quote
\r	Carriage return		

Table 3: Escape sequences

Example 5: Tabulated output

The following Java code shows an example of how escape sequences work.

```
public class EscapeSeq
{
        public static void main(String [] args)
        {
            System.out.print("\n");
```

(continued on next page)

(continued from previous page)

```
            System.out.println("Name\tHeight\tGender");
            System.out.println("----------------------");
            System.out.println("Anna\t5\'4\"\tF");
            System.out.println("Artit\t6\'2\"\tM");
            System.out.println("Bina\t5\'7\"\tF");
        }
}
```

In this example, \n is used for entering a new line and \t is used for inserting the tab character. The use of escape sequences is useful when the output to the screen needs to be formatted nicely.

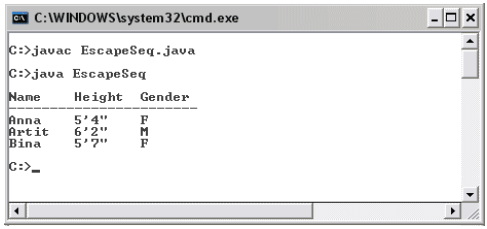


Figure 20: Demonstration of some usages of escape sequences

Variable at a Glance

Variables are symbolic names of memory locations. They are used for storing values used in programs. We will discuss Java types of values and variables later in this book. In many programming languages including Java, before a variable can be used, it has to be declared so that its name is known and proper space in memory is allocated. Syntax used for declaring variables can be observed from Figure 21.

```
        int x;
        double y;
        String myText;
```

Figure 21: Declaration of variables

On the first line, a variable **x** is created for storing an **int** (integer) value. On the second line, a variable **y** is created for storing a **double** (double-precision floating point) value. On the last line, a variable **myText** is created for storing a reference to *String* (a Java standard class representing double quoted text) object. The name **x**, **y**, and **myText** are Java identifiers. **int** and **double** are Java keywords. *String* is the name of a Java class.

Variables are normally used with the *assignment operator* (=), which assign the value on the right to the variable on the left as can be observed in Figure 22

```
x = 3;
y = 6.5;
myText = "Java Programming";

int z;
z = x;
```

Figure 22: Assigning values to variables

On the first three lines, values are assigned to the three variables according to their types. On the last two lines, a variable **z** is declared and then assigned the value of the variable **x**.

Declaration and value assignment (initialization) could be done in the same statement as can be observed in Figure 23

```
int i = 1;
double f = 0.0;
String mySecondText = "Java is fun.";
```

Figure 23: Declaring and assigning values to variables in single statements

Naming Rules and Styles

There are certain rules for the naming of Java identifiers. Valid Java identifier must be consistent with the following rules.

- An identifier cannot be a Java reserve word.
- An identifier must begin with an alphabetic letter, underscore (_), or a dollar sign ($).
- If there are any characters subsequent to the first one, those characters must be alphabetic letters, digits, underscores (_), or dollar signs ($).
- Whitespace cannot be used in a valid identifier.
- An identifier must not be longer than 65,535 characters.

Also, there are certain styles that programmers widely used in naming variables, classes and methods in Java. Here are some of them.

- Use lowercase letter for the first character of variables' and methods' names.
- Use uppercase letter for the first character of class names.
- Use meaningful names.
- Compound words or short phrases are fine, but use uppercase letter for the first character of the words subsequent to the first. Do not use underscore to separate words.
- Use uppercase letter for all characters in a constant. Use underscore to separate words.
- Apart from the mentioned cases, always use lowercase letter.
- Use verbs for methods' names.

Here are some examples for good Java identifiers.

- Variables: **`height`**, **`speed`**, **`filename`**, **`tempInCelcius`**, **`incomingMsg`**, **`textToShow`**.
- Constant: **`SOUND_SPEED`**, **`KM_PER_MILE`**, **`BLOCK_SIZE`**.
- Class names: **`Account`**, **`DictionaryItem`**, **`FileUtility`**, **`Article`**.
- Method names: **`locate`**, **`sortItem`**, **`findMinValue`**, **`checkForError`**.

Not following these styles does not mean breaking the rules, but it is always good to be in style!

Statements and Expressions

An *expression* is a value, a variable, a method, or one of their combinations that can be evaluated to a value. Each line in Figure 24 is an expression.

```
3.857
a + b – 10
8 >= x
p || q
"go"
Math.sqrt(2)
squareRootTwo = Math.sqrt(2)
```

Figure 24: Java expressions

A *statement* is any complete sentence that causes some action to occur. A valid Java statement must end with a semicolon. Each line in Figure 25 is a Java statement.

```
int k;
int j = 10;
double d1, d2, d3;
k = a + b – 10;
boolean p = ( a >= b );
System.out.print(“go”);
squareRootTwo = Math.sqrt(2);
```

Figure 25: Java statements

Each of statement in Figure 26 is not a valid Java statement since it either does not cause any action to occur or is syntactically incorrect.

```
i;
2 <= 3;
1 + 2 + 3 + 4;
“go”;
Math.sqrt(2);
```

Figure 26: Invalid Java statements

Simple Calculation

Numeric values can be used in calculation using arithmetic operators, such as add (+), subtract (-), multiply (*), divide (/), and modulo (%). An *assignment operator* (=) is used to assign the result obtained from the calculation to a variable. Parentheses are used to define the order of the calculation.

Example 6: Simple calculation and adaptation

The following program computes and prints out the average of the integers from 1 to 10.

```
public class  AverageDemo                                          1
{                                                                  2
       public static void main(String[] args)                     3
       {                                                           4
           double avg, sum;                                        5
           sum = 1.0+2.0+3.0+4.0+5.0+6.0+7.0+8.0+9.0+10.0;         6
           avg = sum/10;                                           7
           System.out.println(avg);                                8
       }                                                           9
}                                                                  10
```

In this example, the statement on line 5 declares two variables that are used to stored floating point numbers. On line 6, the values from 1.0 to 10.0 are summed together using the + operator and the resulting value is assigned to **`sum`**. On line 7, the value of **`sum`** is divided by 10 which to obtain their average. The statement on line 8 just prints the result on screen.

Here, let's look at how we can adapt the *AverageDemo* program a little in order for the new program to calculate the sum of squares of the integer values of 1.0 to 10.0

```
public class  SumSquareDemo                                        1
{                                                                  2
       public static void main(String[] args)                     3
       {                                                           4
           double sumSquare;                                       5
           sumSquare = 1.0*1.0+2.0*2.0+3.0*3.0                     6
```

(continued on next page)

(continued from previous page)

```
                        +4.0*4.0+5.0*5.0+6.0*6.0+7.0*7.0          7
                        +8.0*8.0+9.0*9.0+10.0*10.0;               8
            System.out.println(sumSquare);                        9
        }                                                         10
}                                                                 11
```

A key modification made in this example compared to the previous one is the statement calculating the main result of the program which is the sum of squares of 1.0 to 10.0. The square of each number is found by using the * operator with both operands being that number in order to multiply the number, which consequently results in its square. Then, multiple + operators are used to sum them together.

This example shows that sometimes there is no need to be an expert of the language (just yet!) to modify an existing source code for it to work as we desire.

More operators will be presented later in the next chapter.

Representing Algorithms Using Flowcharts

To create a computer program that works, one need not only the knowledge about the rules and syntaxes of a programming language but also a procedure or a process that is used to accomplish the objectives of that program. Such a procedure is called an *algorithm*. Usually, before creating a computer program, an algorithm is developed based on the objective of the program before the source code is written. An algorithm could be as simple as a single command. More often than not, they are more complex.

Representing an algorithm using diagrams is useful both in designing an algorithm for a complicate task and for expressing the algorithm to other people. More than one ways of creating such diagrams have been created and standardized. One of the simplest ways is to use a *flowchart*. Although representing an algorithm using a flowchart might not be an

ideal way for some situations, it is the most intuitive and should be sufficient for beginners of computer programming.

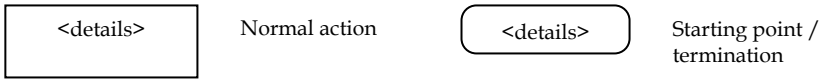


Figure 27: Symbols in a flowchart used for regular actions, starting points and process termination

A flowchart needs a starting point of the program it represents and one or more terminations. Steps or commands involved in the program are represented using rectangles containing textual descriptions of the corresponding steps or commands. These steps as well as the starting and terminating points are connected together with line segments in which arrows are used for determining the order of these steps. Shapes are typically placed from the top of the chart to the bottom according to their orders. The starting and terminating points are represented using round-corner rectangles. Different shapes apart from the two shapes already mentioned are defined so that they imply some specific meanings.

The following flowchart shows an algorithm of a computer program that prints out the average of the integers from 1 to 10 (which is the Java program showed in Example 6).

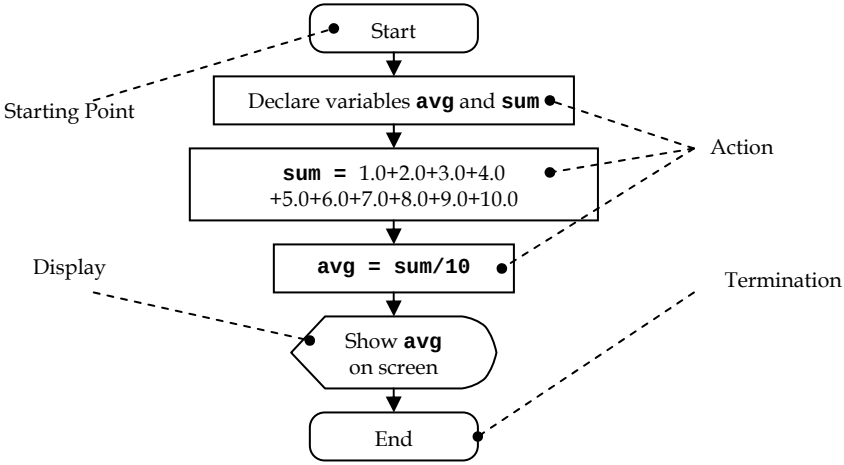


Figure 28: The flowchart of the *AverageDemo* program

In the flowchart above, the shape located just before the termination is used to represent a step involving displaying an output on to the screen.

Manual Input

Skewed rectangles like the one in Figure 29 are used for steps which involve manual input such as received user's inputs from keyboards. Detailed operations will be described as needed inside the shape.

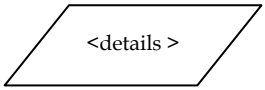


Figure 29: Symbol used in a flowchart for manual input

The following flowchart is another example where the program prompts the user to input his/her name and then prints a greeting message containing the input name on screen.

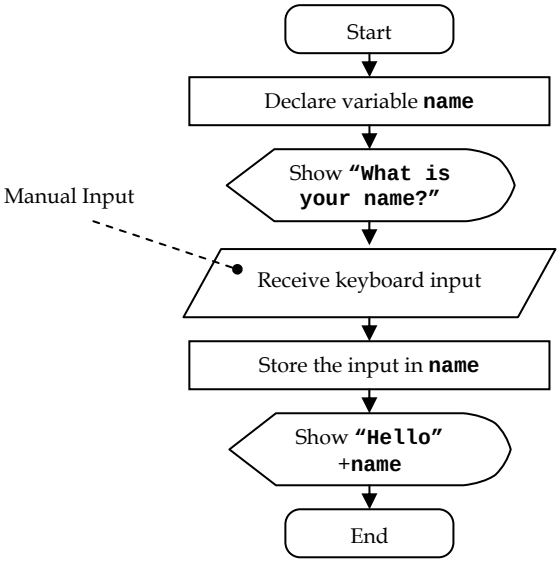
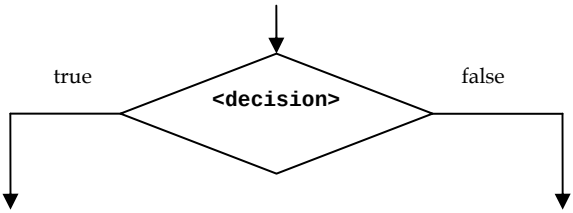


Figure 30: The flowchart of a program greeting a person whose name is manually input via keyboard

Decisions and Iterations

A procedure that always follows a single path every time the program is run might not be very attractive and is hardly useful. Usually, decisions on which steps to be executed have to be made based on some specified



conditions. The symbol used in this case can be shown in Figure 31

Figure 31: Symbol used in a flowchart for decisions

Example 7: An algorithm to find absolute values

Consider the following example. The program shown receives an integer from the user via keyboards and shows the absolute value of that integer using the logic "If an integer, *n*, is non-negative, then its absolute value is just *n*. Otherwise; its absolute value is -*n*". A rhombus specified with its corresponding conditional statement is used for representing a decision step. If the statement is **`true`**, the flow of the program follows the line labeled **`true`**. If it is **`false`**, the flow follows the other path.

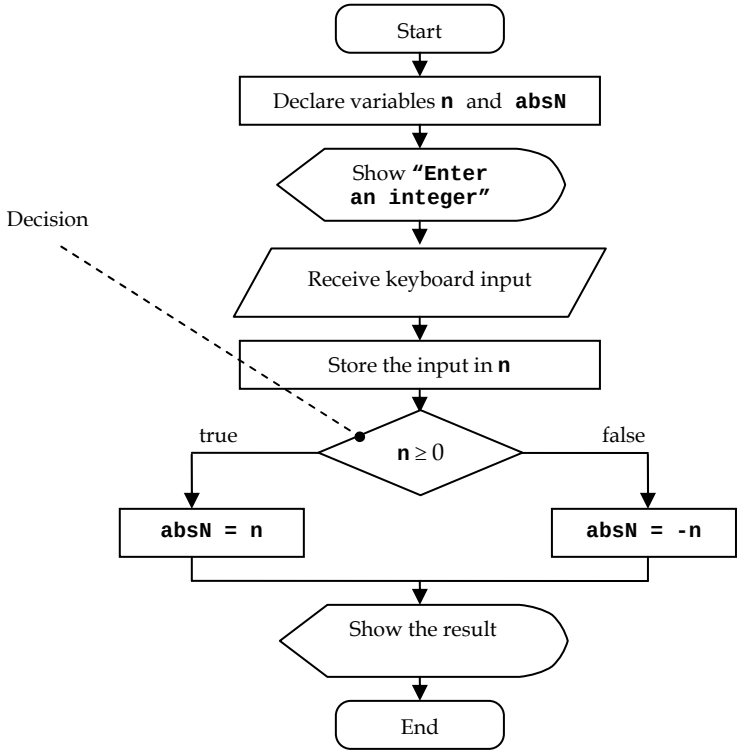


Figure 32: Flowchart of program calculating the absolute value of a manually input integer value

Conditional statements in decision steps might not necessary result in **`boolean`** (**`true`** or **`false`**) values. A decision step could result in more than two paths, given that each path is labeled with the value(s) of the conditional statement that makes the program follow that path.

Decision steps are also useful in developing algorithms in which some portions of them are repeated for a certain number of iterations or until some conditions are met.

Example 8: Factorial function algorithm

The following flowchart demonstrates a program that finds and prints out the value of *n*!, where *n* is a non-negative integer input via keyboards. Note that *n*! is defined as:

$$n! = \begin{cases} n(n-1)(n-2)...1 & n > 1 \\ 1 & n = 0,1 \end{cases}$$

Note that the circular shapes labeled with the same number (which is 1 in this case) connect two parts of the flowchart together.

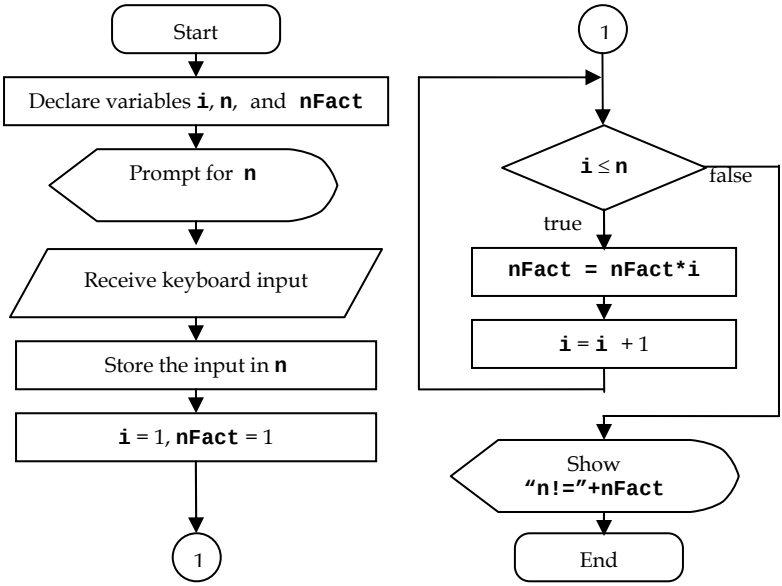


Figure 33: Flowchart of program calculating the factorial of a manually-input integer value

In this example, a loop is used. The value stored in `i` is used for counting the number of iterations that have been performed. From the flowchart, we can see that `i` is initialized with 1, and as long as the value of `i` does not exceed the input `n`, the flow of the program will keep going back to the point just prior to the decision step. This makes the value of `nFact` be updated with the product of the original value of itself and the iteration number stored in `i`. Note that `i` has to be updated in a way so that the loop will eventually be exited.

To actually write a Java program that performs the process in the above flowchart, you will need to know more on rules and syntaxes of the language, which will be covered in later chapters.

Subroutines

A *subroutine* is a portion of code within a larger program or a sub-process within a bigger process. In creating a computer program, specific tasks that might be performed many times or are relatively independent of the rest of the programs are usually defined as subroutines. In Java, subroutines are supported using methods briefly mentioned earlier. More details concerning methods will be discussed in later chapters. The symbol used for subroutines is shown in Figure 34.

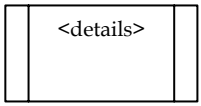


Figure 34: Symbols used in a flowchart for a subroutine

Example 9: Combination function algorithm

The following flowchart shows an example of how to represent subroutines. The program finds the value of the combination function, *c(n,k)*, where *n* and *k* are integer values taken as the inputs to the program. The function is defined as:

$$c(n,k) = \frac{n!}{(n-k)!k!}$$

(Note that this function is used for finding the number of subsets with *k* elements where each element is taken from a set with *n* elements.)

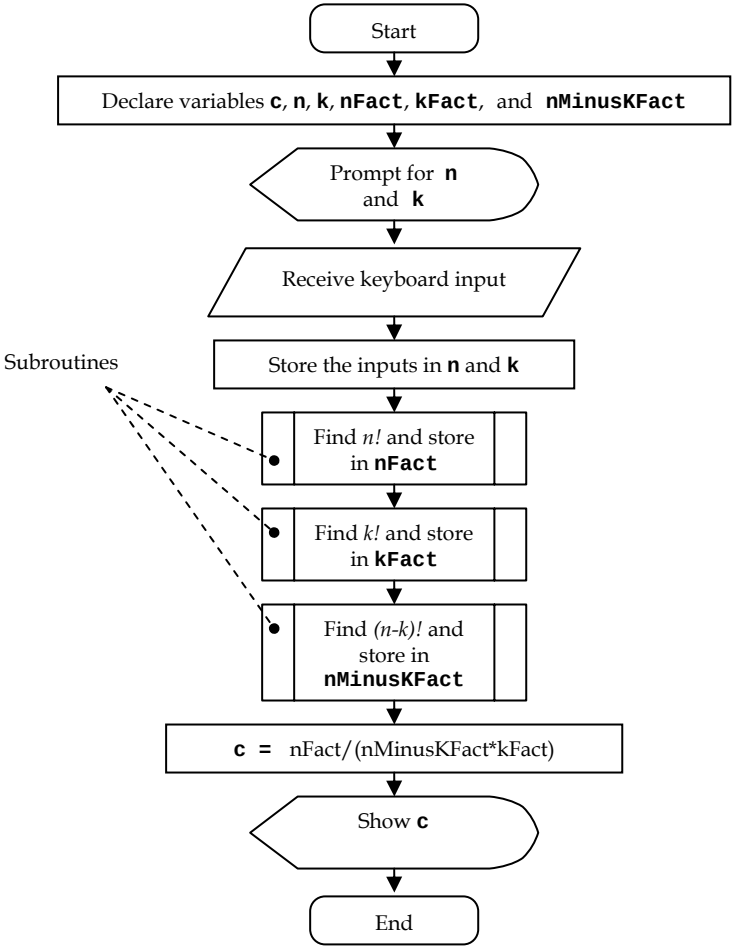


Figure 35: Flowchart of a program calculating a combinatorial function

The sub-process used in finding the value of *n!* is defined in a subroutine whose detailed steps could be described in another flowchart such as the one in the previous example. Notice the symbol used for representing subroutines.

More Shapes

There are more shapes used and adopted in representing processes or algorithms using flowcharts. Here are some more examples of the shapes.

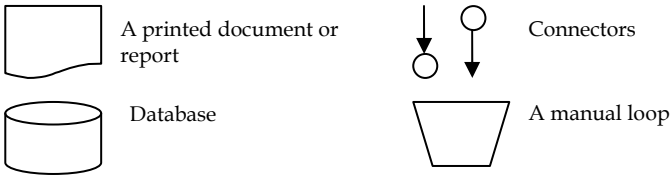


Figure 36: Some of other symbols used in a flowchart

More details on these shapes or other shapes and symbols not mentioned here are intentionally omitted. Although using the proper notation is important in communicating algorithms with other people, our primary concern here is to be able to plan the steps required in creating a program for accomplishing some tasks with the assist of flowcharts.

Example 10: An algorithm for finding the biggest of three inputs

Suppose we would like to create a simple program that lets the user input three numeric values via keyboards and then shows the biggest value among the three inputs, the simplest algorithm might be just performing a comparison to find the maximum value between a pair of the inputs first and then comparing whether the maximum of the two is bigger than the other input. This way, we could use three variables to hold the value of the three inputs and another variable, `m`, to store the most up-to-date maximum value as we do the comparisons. Once we finish the two comparisons mentioned, the value in `m` should be shown on screen. The mentioned process could be described using a flowchart as the following.

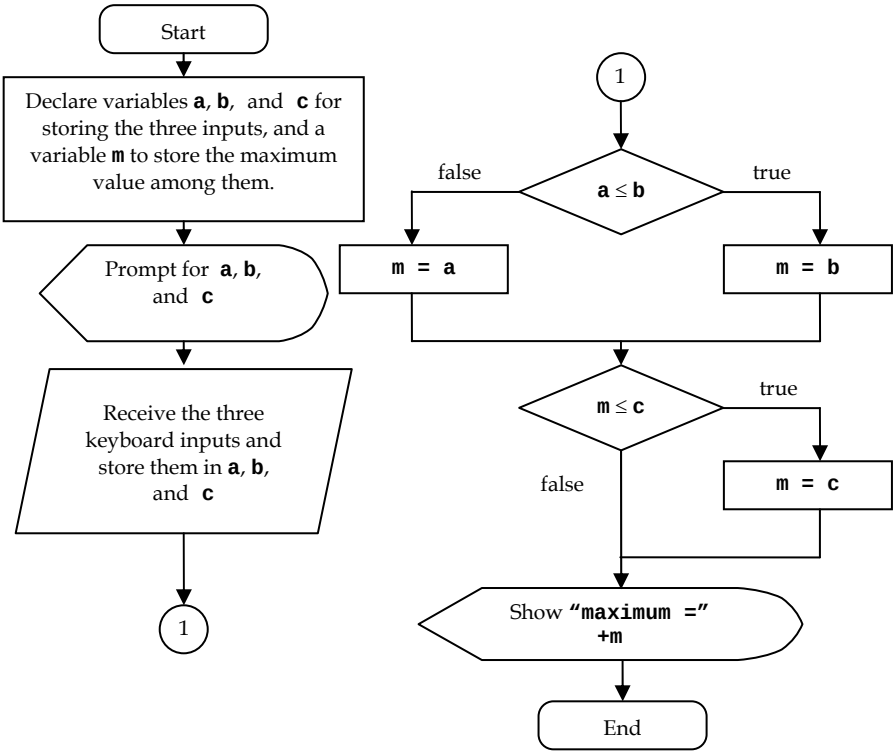


Figure 37: Flowchart of a program finding the biggest of the three manually-input values

---

Exercise

1. What are the differences between machine languages and high-level programming languages?
2. Explain why Java is said to be a platform-independent language.
3. Describe the benefit(s) of using proper indentation.
4. Write a complete Java program that shows your first name and last name on screen.

5. Write a complete Java program that shows the pattern similar to the following.

```
*******      *******
      *      *     *
      *******      *******
```

6. Change this source code as little as possible, so that it gets compiled successfully?

```
public class Ex2_6
{
    public static void main(String [] args)

        System.out.println("What's wrong with me?)
    }
}
```

7. Show how to use method print() once in order to print the following message on screen.

```
He called Clancy at headquarters and said:
"There's a family of ducks walkin' down the street!"
```

8. Which of the following identifiers are not valid?

google	$money	company_motto	java.org
12inches	money$	X-Ray	CC
Item3	$$money	!alert_sign	entry point
public	main	Class	_piece
JavaApplet	fillForms	"Gorilla"	_1234

9. List every keywords and identifiers appeared in the class definition of **`AverageDemo`**.

10. Modify the source code of the class **`AverageDemo`** so that the program shows both the summation and the average of the integer 1 to 5 as the following. Also draw a flowchart for the program.

```
The sum of 1 to 5 is 15.0
The average of 1 to 5 is 3.0
```

11. Draw a flowchart describing a process that a bank customer could use in making cash withdrawal from an ATM machine.
12. Read the following recipe and draw a corresponding flowchart.

Shrimp Scampi Bake

- Preheat oven to 450 degrees F (230 degrees C).
- In a small saucepan over medium heat, combine the butter, mustard, lemon juice, garlic, and parsley. When the butter melts completely, remove from heat.
- Arrange shrimp in a shallow baking dish. Pour the butter mixture over the shrimp.
- Bake in preheated oven for 12 to 15 minutes or until the shrimp are pink and opaque.

Figure 38: A recipe for shrimp scampi bake

Chapter 3: Working with Data

Objectives

Readers should

- Be familiar with the eight primitive data types and non-primitive data types.
- Understand memory allocations and value assignments of variables.
- Be able to use operators and some methods to do some computations.

Strong Data Typing

Programming languages differ in many aspects. One aspect is whether the languages place strict rules on identifying types of every piece of data involved in the program. A programming language is called a *strong data typing* language if the language needs the instructions in the programs to clearly state the types of all data unless the data type can be unambiguously implied. In contrary, if a programming language allows creation of data whose type cannot be determined or it allows data to be used as one type in one place and as another type at a different place without any explicit transformation, the language is not a strong data typing language.

Java as well as languages in the C-family (E.g. C, C++, C#) are strong data typing languages, while some examples of popular programming languages that are not strong data typing are Visual Basic and PHP.

Although writing a program without having to worry about data types might seem tempting to programmers, strong data typing lets the compiler catches possible errors more often and it also enable programming languages to carry out more fancy features. Interested readers can consult [Seb2009] for more advanced topic relating features of programming languages.

Since Java is strongly typed, we have to learn about available data types in Java and how to process data with concerns over their types in this chapter. Conversions among data and variables of different data typed will be discussed in the next chapter.

Data Types in Java

There are 2 main types of data in Java.

1. Primitive data types
2. Classes

Primitive data types are data types considered basic to Java. They cannot be added or changed. There are eight primitive data types in Java including 4 types for integer values, 2 types for floating-point values, 1 type for characters, and 1 type for logical values (true/false).

Value type	Primitive data types
Integer (E.g.: 5, -1, 0, etc.)	byte, short, int, long
Floating-point (E.g.: 1.0, 9.75, -3.06, etc.)	float, double
Character (E.g.: 'a', 'ก', '@', '4', etc.)	char
Logical (true/false)	boolean

Table 4: Primitive data types in Java

Classes are more complex data types. There are classes that are standard to Java such as *String*, *Rectangle*, etc. Also, new classes can be defined by programmers when needed.

Primitive Data Type for Integers

There are four primitive data types for integer values, including **`byte`**, **`short`**, **`int`**, and **`long`**. They are different in the sizes of the space they occupied in memory. The reason why Java has multiple integer (as well as, floating-point) types is to allow programmer to access all the types

support natively by various computer hardware and let the program works efficiently.

Recall that a space of 1 byte (8 bits) can store 2^8 different things.

A **`byte`** occupies 1 byte of memory. It can represent one of 2^8 (256) integers in the range -128, -127, …, 0, 1, …, 126, 127.

A **`short`** occupies 2 bytes of memory. It can represent one of 2^{16} (65,536) integers in the range -32,678, …, 0, 1, …, 32,767.

An **`int`** occupies 4 bytes of memory. It can represent one of 2^{32} (4,294,967,296) integers in the range -2,147,483,648, …, 0, 1, …, 2,147,483,647.

A **`long`** occupies 8 bytes of memory. It can represent one of 2^{64} integers in the range -2^{63}, …, 0, 1, …, $2^{63}-1$.

The default primitive type for integer value is **`int`**.

Choosing the data type for an integer value according to its range leads to efficient data processing as well as memory usage. However, for beginners focusing on writing small application programs, it is perfectly fine to handle all integer value with the **`int`** data type, unless the values are beyond the value range of **`int`**.

Primitive Types for Floating-Point Values

There are two primitive data types for floating point values. They are **`float`**, and **`double`**. They are different in their sizes and precisions. The default primitive type for integer value is **`double`**.

A **`float`** occupies 4 bytes of memory. Its range is from -3.4×10^{38} to 3.4×10^{38}, with typical precision of 6-9 decimal points.

A **`double`** occupies 8 bytes of memory. Its range is from -1.8×10^{308} to 1.8×10^{308}, with typical precision of 15-17 decimal points.

There are two ways to write floating-point values. They can be written in either decimal or scientific notation. 123.45, 0.001, 1.0, 8.357, 1., and .9 are in decimal notation. 1.2345**E**2, 10**E**-4, 1**E**0, 0.8375**E**1, 1000**E**-3, and 9.0**E**-1 are in scientific notation representing 1.2345×10^{2}, 10×10^{-4}, 1×10^{0}, 0.8375×10^{1}, 1000×10^{-3}, and 9.0×10^{-1} respectively. The character **E** can also be replaced with **e**.

It is usually okay to use **double** for all of the floating point values in the program.

Primitive Type for Characters

The primitive data type for characters is **char**. A character is represented with single quotes. For example the character for the English alphabet Q is represented using 'Q'.

Characters include alphabets in different languages ('a', 'b', 'ก'), numbers in different languages ('1', '2', '๑'), symbols ('%', '#', '{'), and escape sequences ('\t', '\n', '\"');

In fact, the underlying representation of a **char** is an integer in the Unicode character set coding. Characters are encoded in 2 bytes using the Unicode character set coding.

Character	Unicode encoding
'A'	65
'B'	66
'a'	97
'b'	98
'1'	49
'#'	35

Table 5: Some examples of characters with their corresponding unicode encodings

Unicode character set encoding guarantees that '0' < '1' < '2' < … < '9', 'a' < 'b' < 'c' < … < 'z', and 'A' < 'B' < 'C' < … < 'Z'. Also,

'0'+1 is '1', '1'+2 is '3', …, '8'+1 is '9',

'a'+1 is 'b', 'b'+1 is 'c', …, 'y'+1 is 'z', and

'A'+1 is 'B', 'B'+1 is 'C', …, 'Y'+1 is 'Z'.

Primitive Type for Logical Values

There are only two different logical values, which are **`true`** and **`false`**. A primitive type called **`boolean`** is used for logical values. In Java, 0 does not mean **`false`** and 1 does not mean **`true`**. Logical values and numeric values cannot be interchangeably.

String

String is an example of a data type that is not a primitive data type. String is a standard class in Java. It is used for representing a sequence of zero or more characters. Double quotes are used to designate the *String* type. For example, "Wow!" is a data of *String* type. Be aware that "100" is not the same as the integer 100, and "Q" is not the same as the character 'Q'.

Pointers

In some programming languages such as C and C++, there is another important data type called *pointer*. Data of the type pointer, or simply called pointers, are memory locations or addresses of other data or instructions. The use of pointers allows accessing data stored in specific memory locations directly. In Java, pointers are not available. Most functionality traditionally obtained by the use of pointers is available with some other mechanisms in Java.

Assigning Data to Variables

Data can be assigned to or stored in variables using the assignment operator (=). Like data, variables have types. A variable can only store a value that has the same data type as itself. Assigning values whose data types are different from the variables to which they are assigned must involve data type conversions. Such conversions can be done either implicitly or explicitly using *cast operators*. More discussion on this will be brought up in the next chapter.

Variable types are specified when the variables are declared, using the name of the desired data types followed by the name of the variables. Do not forget semicolons at the end.

For each variable whose type is one of the primitive data types, memory space of the size corresponding to its type is allocated. The allocated space is used for storing the value of that type. However, it works differently for a variable of a non-primitive type. The memory space allocated for such a variable is for what is called a *reference*. When assigned with a value, the reference points, or refers, to the memory location that actually stores that value.

We have discussed briefly about how to declare variables and assign values to them in the last chapter. Here, we revisit it again with another example. This time, attention should be paid on the data types.

To better illustrate the memory allocation and value assignments of variables, we add to the following example code segment with some illustration of the allocated memory. Here, we represent a variable by a box captioned with its name. When a value is assigned to a variable, we write the value inside the box associated with that variable.

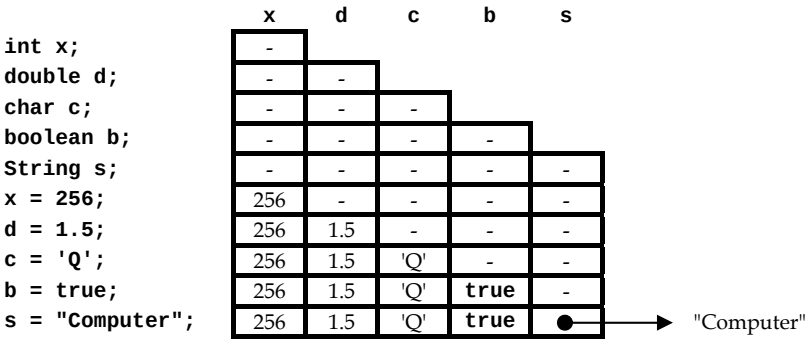


Figure 39: Storing values into variables

On the first four lines of the above code segment, a variable **x** of type **int**, a variable **d** of type **double**, a variable **c** of type **char**, and a variable **b** of type **boolean** are created and allocated with memory space of the size according to the type. On the fifth line, **String s** is the declaration of a variable **s** of *String* type. On this line, a reference to *String* is created but it has not referred to anything yet. Variables are assigned with values of the corresponding data types on the last five lines. For the last line, the reference in **s** is made to point to the *String* "Computer" located somewhere else in the memory.

Figure 40 shows an example of *String* assignment between two variables.

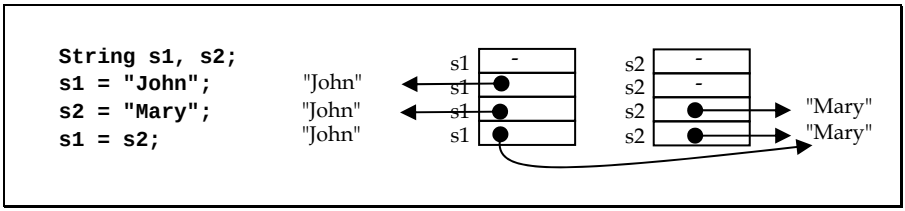


Figure 40: Reassigning a String variable

The next figure contains another code segment showing what we can do. The discussion comes after the figure.

```
int x,y = 8;
double z = 0.6, w;
double k=3.0;
w = k;
x = y;
```

Figure 41: Various variable declaration and assignment examples

The first two lines show how to declare two variables of the same type in one statement. Also, they show how to assign values to variables at the same time as the variables are declared. Note that, the integer value 8 is only assigned to **y**, not **x**, on the first line. On second line from the bottom, the **double** value stored in **k** is assigned to the variable **w**. On the last line, the **int** value stored in **y** is assigned to the variable **x**.

Figure 42 and Figure 43 show some examples of bad variable declarations and/or assignments.

```
int i = 9                              // no semicolon at the end
int j = 1.0;                           // mismatched data type
boolean done = "false";                // mismatched data type
char input character = 'x';            /* syntax error
                                        or illegal identifier */
Int k = 1;                             // Undefined data type
double k; m = 5e-13;                   // undefined variable m
char class = 'A';                      /* use a reserve word
                                        as an identifier */
String s1 = 'W';                       // mismatched data type
string s2;                             // Undefined data type
```

Figure 42: Invalid variable declarations and assignments

```
int i, j;
int k = 2;
int i = 6;     // declaration of redundant variable i
int j = 8;     // declaration of redundant variable j
```

Figure 43: Redundantly declaring variables

Final Variables

Sometimes, we want to store a value that cannot or will not be changed as long as the program has not terminated in a variable. The variable with such an unchangeable value is referred to as a *final variable*. The keyword **`final`** is used when a variable is declared so that the value of that variable cannot be changed once it is initialized, or assigned for the first time. Programmers usually use all-uppercase letters for the identifiers of final variables, and use underscore (_) to separate words in the identifiers (E.g.: YOUNG_S_MODULUS, SPEED_OF_LIGHT). Attempting to change the value of a final variable after a value has been assigned results in an error.

For example:

```
// Declaration and assignment in a single statement
final double G = 6.67e-11;

// Declare first and then assign the value only once
final double SPEED_OF_SOUND;
SPEED_OF_SOUND = 349.5; // Speed of sound at 30 degree celcius
```

The following code segment will produce an error due to the assignment statement on line 8. Observe the explanation of the error given by the compiler in the output screen in Figure 44.

```
public class FinalVariableError {                                    1
    public static void main(String [] args){                        2
        final int C;                                                 3
        int k = 6;                                                   4
        C = 80;                                                      5
        C = k + 300;                                                 6
    }                                                                7
}                                                                    8
```

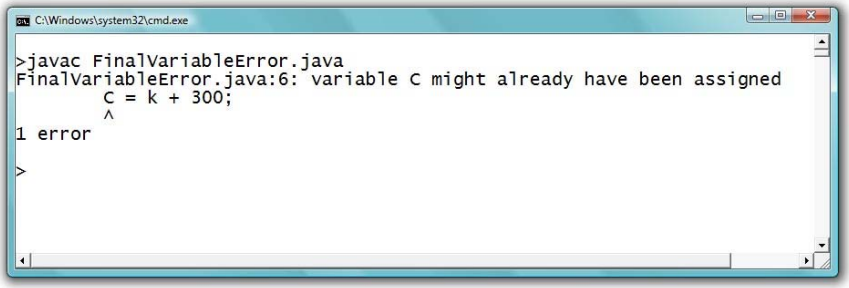


Figure 44: Assigning a new value to a final variable

Un-initialized Variables

When a variable is declared, a space in the memory is reserved for the size of the type of that variable. However, as long as it is not assigned with a value, the variable does not contain any meaningful value, and it is said that the variable has not been *intitialized*. If the value of an un-initialized variable is used, an error occurs.

The following code segment will produce an error due to the attempt to use an un-initialized variable (which is x) on line 4 .

```
public class UninitVarError {                                    1
    public static void main(String [] args){                    2
        int x, y = 2;                                            3
        System.out.println(x+y);                                 4
    }                                                            5
}                                                                6
```

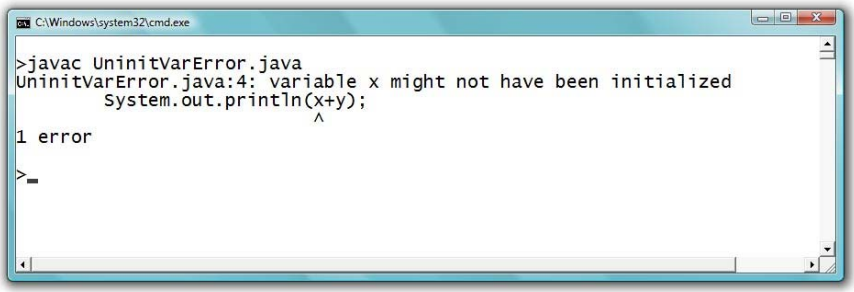


Figure 45: Using an uninitialized variable

Operators

Values can be manipulated using built-in arithmetic and logic operators (such as +, -, *, /, %), or using available methods (such *as abs(), sqrt(), exp(), floor(), log(), getTime()*). Many methods are defined in some standard classes such as the *Math* class. Although you are expected to be able to use the methods that have previously seen in this class, such as *print()* and *println()*, we will defer the detailed discussion on using methods for now. In this chapter, we will mostly discuss about using built-in operators and some small number of selected methods to manipulate data.

Arithmetic operators that you should know include addition (+), subtraction (-), multiplication (*), division (/), modulo (%), for which $a\%b$ returns the remainder of $a \div b$, and negation (-).

Logic operators that you should know is *logical AND* (&&), *logical OR* (||), and *logical negation* (!).

Portions of code that are either constant values, variables, methods, or any combinations of the mentioned items from which some values can be computed are called *expressions*.

Logic operator	Usage
&&	**a&&b** yields true if and only if both **a** and **b** are true.
||	**a||b** yields false if and only if both **a** and **b** are false.
!	**!a** yields the opposite logical value of **a**.

Table 6: Description of logic operators

Operators that require two operands are called *binary operators*, while operators that require only one operand are called *unary operators*.

Parentheses, (), are called *grouping operators*. They indicate the portions of the calculation that have to be done first.

Values can be compared using relational *equality operators*, == and !=, which return either one of the boolean values depending on the value of the operand. **a==b** yields true if and only if **a** and **b** have the same value. **a!=b** yields true if and only if **a** and **b** have different value. Comparison can also be done using <, >, <=, and >=.

Below are two examples showing how values of variables change due to value assignments and the use of some operators.

```
int i = 2, j, k;
j = 3;
k = i + j;
i = i + k;
```

i	j	k
2	-	-
2	3	-
2	3	5
2	3	7

Figure 46: Operating on int variables

```
boolean p = true;
boolean q = false, r = true, s;
s = p && q;
s = s || r;
p = (s == q);
```

p	q	r	s
true			
true	false	true	-
true	false	true	false
true	false	true	true
false	false	true	true

Figure 47: Operating on boolean variables

String and the addition operator (+)

The addition operator (+) can be used to concatenate two Strings together. For example, "Comp"+ "uter." will results in "Computer". The concatenation can also be used upon Strings that are stored in variables.

The following program shows an example of concatenating two String objects referred to by two variables and assigning the new String to another String variable.

```
public class StringConcat1 {                                    1
    public static void main(String [] args){                    2
        String s1 = "Computer", s2 = "ized", s3;                 3
        s3 = s1 + s2;                                            4
    }                                                            5
}                                                                6
```

Figure 48: String concatenation with + operator

The variable **s3** contains the String "Computerized" since it is the result of concatenating **s1** and **s2** with the + operator on line 4.

Whenever one of the operands of the addition operator is a String, the operator performs the String concatenation. Thus, if a String is added with values of different data types, the compiler will try to convert the values that are not String to String automatically. Good news is that the automatic conversion usually returns the String that makes very much sense! E.g.: numeric values will be converted to the Strings whose contents are consistent with the original numbers.

Example 11: Using + to concatenate multiple strings together

Observe this source code and its output.

```
public class StringConcat2                                      1
{                                                               2
       public static void main(String[] args)                   3
       {                                                        4
           double distance, speed;                              5
           distance = 2500; // meters                           6
```

(continued on next page)

(continued from previous page)

```
            speed = 80; // km per hour                                  7
            System.out.print("It would take a car”);                    8
            System.out.print(“running at "+speed+" km/h ");             9
            System.out.print((distance/1000/speed)*60*60+" sec.”);      10
            System.out.print(“to travel ”);                             11
            System.out.println(distance/1000+" km.");                   12
        }                                                               13
}                                                                       14
```

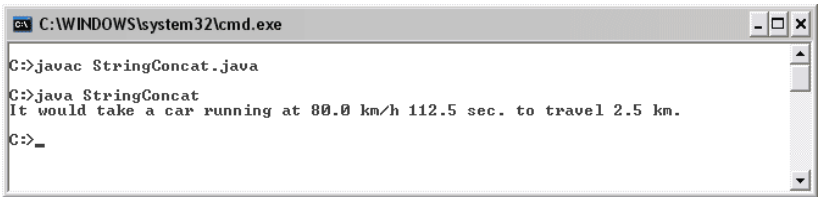


Figure 49: Using + operators with String and other data types

Consider the input parameter to *print()* on line 9, the input to the method is an expression that uses two + operators on operands which are Strings as well as a number. The number in `speed` is converted to `“80.0”` prior to String concatenation according to the principle mentioned just above. Same things are applied with the statements on line 10 and on line 12.

Math methods

Among the first groups of problems that we usually write computer programs to solve are performing mathematics calculation. Here are some example methods that provide useful mathematic functions that we might come across frequently.

`Math.abs(<a numeric value>);`
: returns the absolute value of the input value.

`Math.round(<a numeric value>);`
: returns the integer nearest to the input value.

`Math.ceil(<a numeric value>);`

returns the smallest integer that is bigger than or equal to the input value.

`Math.floor(<a numeric value>);`
returns the biggest integer that is smaller than or equal to the input value.

`Math.exp(<a numeric value>);`
returns the exponential of the input value.

`Math.max(<a numeric value>,<a numeric value>);`
returns the bigger between the two input values.

`Math.min(<a numeric value>,<a numeric value>);`
returns the smaller between the two input values.

`Math.pow(<a numeric value>,<a numeric value>);`
returns the value of the first value raised to the power of the second value.

`Math.sqrt(<a numeric value>);`
returns the square root of the input value.

`Math.sin(<a numeric value>);`
returns the trigonometric sine value of the input value in radian.

`Math.cos(<a numeric value>);`
returns the trigonometric cosine value of the input value in radian.

`Math.tan(<a numeric value>);`
returns the trigonometric tangent value of the input value in radian.

Example 12: Mathematic methods

The source code below demonstrates how to perform the calculation of many mathematic functions. Readers should observe the result shown on each line of the output in comparison to the statements presented on the corresponding line of the source code to see that the methods work in the fashion described above.

```
public class MathTest                                                    1
{                                                                        2
    public static void main(String[] args)                              3
    {                                                                    4
      double a = 2.8, b = 3.1, c = 6.0;                                  5
      System.out.println("a+b \t\t= " + (a+b));                         6
      System.out.println("|a| \t\t= " + Math.abs(a));                   7
      System.out.println("round(a) \t= " + Math.round(a));              8
      System.out.println("ceil(a) \t= " + Math.ceil(a));                9
      System.out.println("floor(a) \t= " + Math.floor(a));              10
      System.out.println("exp(a) \t\t= " + Math.exp(a));                11
      System.out.println("max of a and b \t= " + Math.max(a,b));        12
      System.out.println("min of a and b \t= " + Math.min(a,b));        13
      System.out.println("2^c \t\t= "+Math.pow(2,c));                   14
    }                                                                    15
}                                                                        16
```

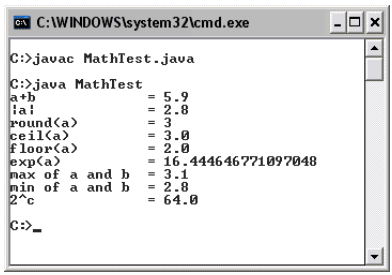


Figure 50: Using + operators with String and other data types

Precedence and Associativity

Consider the following expression.

```
int myNumber = 3 + 2 * 6;
```

This expression is evaluated to 30 if the addition operator is executed first. However, it is 15 if the multiplication operator is executed first. In fact, Java compiler has no problem with such ambiguity. Order of the operators can be determined using *precedence* and *association* rules.

Each operator is assigned a precedence level. Operators with higher precedence levels are executed before ones with lower precedence levels. Associativity is also assigned to operators with the same precedence level. It indicates whether operators to the left or to the right are to be executed first, in the case of equal precedence levels.

Expressions in parentheses () are executed first. In the case of nested parentheses, the expression in the innermost pair is executed first.

Operator	Precedence	Associativity
Grouping operator (())	**17**	Left to right
Unary operator (+, -, !)	**13**	Right to left
Multiplicative operator (*, /, %)	**12**	Left to right
Additive operator (+, -)	**11**	Left to right
Relational ordering (<, >, <=, >=)	**10**	Left to right
Relational equality (==, !=)	**9**	Left to right
Logical and (&&)	**4**	Left to right
Logical or (||)	**3**	Left to right
Assignment (=)	**1**	Right to left

Table 7: Precedence and associativity of some basic operators

Example 13: Applying rules on precedence and associativity (1)

Evaluate the following expression by clearly state the order of operations of all operators according to the precedence and associativity rule.

```
4*2+20/4
```

There are three operators in the above expression. They are *, + and /. The precedence values of * and / are both 12, while the precedence value of + is just 11. Therefore, * and / must be operated prior to +. Since * and / have the same precedence value, we need to look at their associativity which we can see that the one on the left have to be performed first. Therefore, the order of operation from the first to the last is *, / and then +. Consequently, the evaluation of the expression value can take place in steps shown in the figure below, and the resulting value can be shown to be 13.

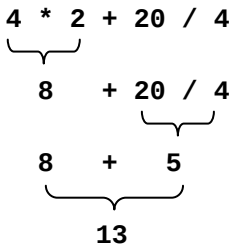


Figure 51: Order of operations in evaluating 4*2+20/4

Example 14: Applying rules on precedence and associativity (2)

Evaluate the following expression.

```
2+2==6-2+0
```

Considering the precedence values of the four operators appearing in the expression, which are + (the leftmost one), ==, –, and + (the rightmost one), we can see that +, and – have the same precedence value of 11 (additive operators) which is higher than the one of ==. Among the three additive operators, we perform the operation from the left to the right according to their associativity. The resulting value of this expression can be evaluated to `true` according to the figure below.

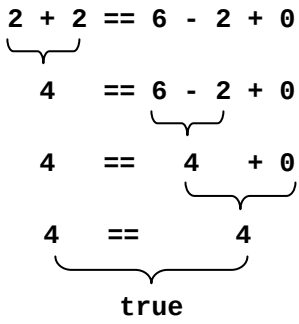


Figure 52: Order of operations in evaluating 2+2==6-2+0

Example 15: Applying rules on precedence and associativity (3)

Evaluate the following expression. Also determine the value of the variable `x` after the expression is evaluated. Assume that the variable `x` has already been properly declared as an `int` variable.

```
(x=3)==(x=+1-2)&&true
```

First, we perform the expression in the left pair of parentheses. The variable `x` is assigned with the `int` value 3 and this is also the resulting value of the expression in this pair of parentheses. Then, the expression `x=+1-2` is evaluated due to the fact that it is in the next pair of parentheses. In this expression, we have the assignment operator = (with the precedence value of 1), the unary positive + (with the precedence value of 13), and the binary operator – (with the precedence value of 13). Based on the comparison of their precedence value, the unary positive is performed first. This operator just indicates the positiveness of its operand. Consequently, the value of the right side of the assignment operator is -1 and then it is assigned to `x`. Therefore, the new value of `x` is -1 which is the value of this pair of parentheses too. The next operator to be performed is the equality operator ==. It compares the values of `(x=3)`and`(x=+1-2)`, which have just been shown that they are not equal. Therefore, the resulting value associated with the action of this operator is the `boolean` value `false`. Finally, the logical AND (&&) is performed

and the final result of the expression in this example is the **boolean** value **false**. Figure 53 illustrates the order of operations described above.

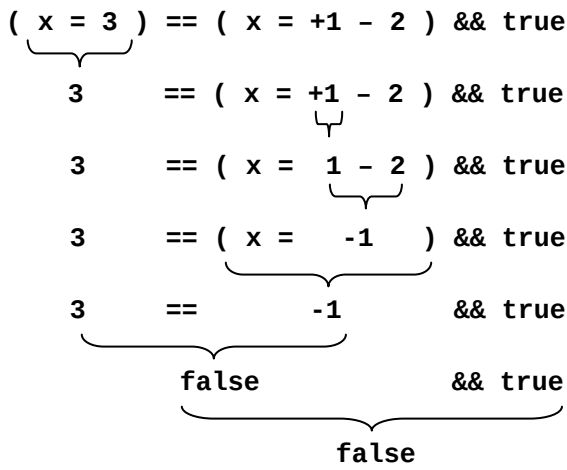


Figure 53: Order of operations in evaluating (x=3)==(x=+1-2)&&true

Example 16: Applying rules on precedence and associativity (4)

Place the grouping operators () into the following expression in order to explicitly determine the order of operations of all operators appearing in the expression. Evaluate the values of every expression involved in steps according to the inserted parentheses.

```
-9.0+5.0*3.0-1.0/0.5 >= 5.0%2.0&&6.0+3.0-9.0==0
```

By considering the precedence values of all operators appearing in the expression above, we can place parentheses into the expression in order to explicitly determine the order of operation and then evaluate the values of each part as shown below.

```
((-9.0)+(5.0*3.0)-(1.0/0.5) >= (5.0%2.0))&&(((6.0+3.0)-9.0)==0)
((-9.0)+15.0     - 2.0      >=  1.0     )&&((     9.0 -9.0)==0)
(            4.0            >=  1.0     )&&(             0 ==0)
(                   true                 )&&(       true       )
true.
```

Let's finish this chapter with a more realistic example of a program that makes use of some mathematic functions from the methods defined in the Math class.

Example 17: Distance in 3-D space

The distance, *d*, between two points in the three-dimensional space (x_1,y_1,z_1) and (x_2,y_2,z_2) can be computed from:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

The program in Figure 54 computes and shows the distance between (2,1,3) and (0,0,6).

```
public class Distance3d                                                    1
{                                                                          2
  public static void main(String[] args)                                  3
  {                                                                        4
    double x1,y1,z1,x2,y2,z2, d;                                           5
    double xDiff, yDiff, zDiff;                                            6
    x1 = 2.0; y1 = 1.0; z1 = 3.0;                                          7
    x2 = 0.0; y2 = 0.0; z2 = 6.0;                                          8
    xDiff = x1-x2;                                                         9
    yDiff = y1-y2;                                                         10
    zDiff = z1-z2;                                                         11
    d = Math.sqrt(xDiff*xDiff+yDiff*yDiff+zDiff*zDiff);                    12
    System.out.print("The distance between");                             13
    System.out.print("("+x1+","+y1+","+z1+") and");                       14
    System.out.println(" ("+x2+","+y2+","+z2+") is "+d+".");               15
  }                                                                        16
}                                                                          17
```

Figure 54: The source code of the Distance3d program

On line 5 and line 6, we declare all variables needed in the program. Then, on line 7 and line 8, we assign specific values into variables associated with the co-ordinates of the two points. To make the statement cleaner, we decide to calculate the different between the two points in each dimension first (on lines 9-11) and store them into the variable **`xDiff`**, **`yDiff`**, and **`zDiff`**. We then multiply each variable that contain the difference with itself to obtain the square of its value before passing their summation as an input to *Math.sqrt()*, on line 12. The result

is assigned to `d`. Finally, the statements on lines 13 to 15 format the result and nicely place them on screen.

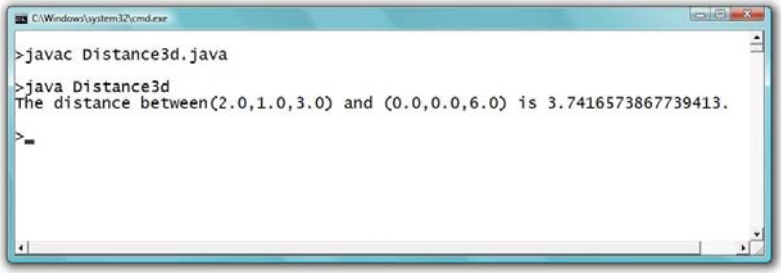


Figure 55: The output of the Distance3d program

Exercise

1. What are the differences between primitive data types and class?
2. How many different things can be represented using *n* bytes?
3. Which of the eight primitive data types can store a numeric value 2^{36}?
4. Give the **`boolean`** values of the following expressions.
    a. 01<2e-1
    b. 8+0.0 >= 8.0
    c. 'a'>'b'
    d. 2+3*2-6 == ((2+3)*2)-6
    e. 'a'>'$' | |'b'<'$'
    f. !(true| |'6'>'#')&&!false
5. Determine the resulting values of **`x`** and **`y`** in the following code.

```
public class  Ex3_5
{
    public static void main(String[] args)
    {
        int x=0,y=0;
        x = y + 1;
        y = x + 1;
    }
}
```

6. Give the reason why the following code will not be compiled successfully.

```
public class Ex3_6
{
    public static void main(String[] args)
    {
        int x, y, z =3;
        y = x;
        z = y;
    }
}
```

7. Write a Java program that calculates and shows these values on screen.
    a. (6+5)(7-3)
    b. $e^{\sqrt{\left|-(6.5)^2+(3.7)^2\right|}}$
    c. $\sqrt[3]{\sin(1.2)}$
    d. The floating point result of $\frac{8}{1+10}$
    e. The biggest integer that is less than 3.75
    f. The smallest integer that is bigger than 3.75
    g. $\sum_{i=0}^{3} i^{(i+1)}$
8. Write a Java program that shows the truth values of the following statements on screen for all possible combination of **p** and **q**. (i.e. (**true**,**true**), (**true**,**false**), (**false**,**true**), and (**false**,**false**))
    a. **p** and **q**
    b. **p** or **q**
    c. either **p** or **q**
    d. either **p** or its negation
9. Modify Distance3D.java so that the distance *d* is calculated from:

$$d = \frac{\sqrt{w_x^{\,2}(x_1 - x_2)^2 + w_y^{\,2}(y_1 - y_2)^2 + w_z^{\,2}(z_1 - z_2)^2}}{\sqrt{w_x^{\,2} + w_y^{\,2} + w_z^{\,2}}}$$

where $w_x = 0.5$, $w_y = 0.3$, and $w_z = 0.2$.

10. Write a Java program that performs the following steps.

    a. Declare two `int` variables named `x` and `y`.
    b. Assign 3 to `x`.
    c. Assign twice the value of `x` to `y`.
    d. Interchange the value of `x` and `y` (without explicitly assign 3 to `y`).
    e. Print the values of both variables on screen.

11. Show a single Java statement that can perform both tasks in step b and c in the last problem.

Chapter 4: More Issues on Data Manipulation

Objectives

Readers should

- Understand how to work with different data types including calculations and conversions.
- Be aware of imprecision in floating point calculations.
- Understand overflow, underflow, `Infinity`, `NaN`, and divided-by-zero exception.
- Be able to use compound assignments correctly.
- Understand and be able to use increment and decrement operators correctly.

Numeric Data Type Specification

If we type numbers without decimal points, such as 1, 0, -8, 99345 etc., the data types of those numbers are all `int` by default. If we type numbers with decimal points, such as 1.0, 0.9, -8.753, 99345.0 etc., the data types are all `double` by default.

At times, we may want to specify a specific data type, apart from the two default types (`int` and `double`), to some numeric values. This can be done by:

- adding an `l` or an `L` behind an integer to make its type `long`.
- adding an `f` or an `F` behind a floating point value to make its type `float`.
- adding a `d` or a `D` behind a floating point value to make its type `double`.

Note that for the type `double`, adding a `d` or a `D` is optional, since `double` is the default type for a floating point value.

All of the following statements assign data to variables whose types are similar to the type of the data.

```
int i = 1;
long l = 1L;
long k = 256l;
float f = 2.0f;
float g = -0.56F;
double d1 = 1.0;
double d2 = 2.0D;
double d3 = 2.0d;
```

Data Type Conversion

As mentioned earlier, data can be assigned to only variables whose types are the same as the data. Failure to do so generally results in a compilation error. However, in some situations we need to assign data to variables of different types. Therefore, data type conversion is needed.

Data type conversion can be done explicitly by using *cast operators*, written as the name of the data type, to which you want to convert your data to, enclosed in parentheses. Cast operators are put in front of the data to be converted. Cast operators have higher precedence than binary arithmetic operators but lower precedence than parentheses and unary operators. That means type conversion by a cast operator is done before +, −, *, /, and %, but after () and unary operators.

Converting floating point numbers to integers will cause the program to discard all of the floating points, no matter how large they are.

Example 18: Data type conversion

Consider the following two programs and their outputs.

```
public class TypeConversionDemo1                                    1
{                                                                   2
        public static void main(String[] args)                     3
        {                                                           4
            int i;                                                  5
            double d = 0.0;                                         6
```

(continued on next page)

(continued from previous page)

```
            i = (int) d;                                    7
            System.out.println("d ="+d);                    8
            System.out.println("i ="+i);                    9
            i = 9;                                          10
            d = (double) i;                                 11
            System.out.println("d ="+d);                    12
            System.out.println("i ="+i);                    13
        }                                                   14
}                                                           15
```

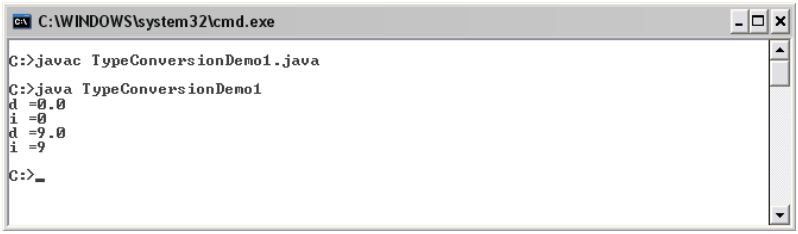


Figure 56: A program demonstrating data type conversion

The cast operator **`(int)`** is used on line 7 for converting the **`double`** value 0.0 stored in the variable **`d`** to the type **`int`**. Then it is assigned to the variable i which is of type **`int`**. The cast operator **`(double)`** was used on line 11 for converting the **`int`** value 9 stored in the variable **`i`** to 9.0. Then, it is assigned to the variable d which is of type **`double`**.

Here is the second program showing explicit type conversions.

```
public class TypeConversionDemo2                            1
{                                                           2
        public static void main(String[] args)             3
        {                                                   4
            int i,j;                                        5
            double d1 = 0.5, d2 = 0.5, d3,d4;               6
            i = (int)d1+(int)8.735f;                        7
            j = (int)(d1+d2);                               8
            System.out.println("i = "+i);                   9
            System.out.println("j = "+j);                   10
            d3 = (double)i-(double)j;                       11
            d4 = (double)(i-j);                             12
            System.out.println("d3= "+d3);                  13
            System.out.println("d4= "+d4);                  14
        }                                                   15
}                                                           16
```

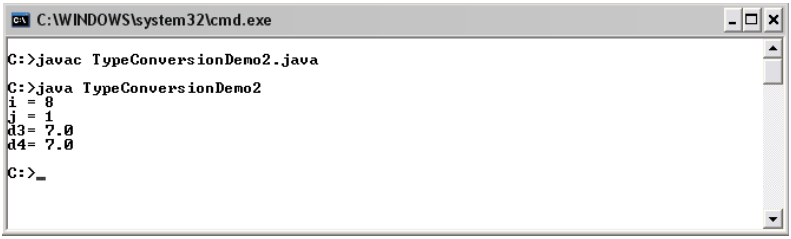


Figure 57: Another program demonstrating data type conversion

On line 7, **`d1`** whose value is a double 0.5 is converted to **`int`** using the **`(int)`** operator. The resulting **`int`** value is 0 since decimal points are truncated. On the same line, the **`float`** value of 8.735f is also converted to the **`int`** value of 8. So, **`i`** equals 0+8 = 8. On line 8, the sum of **`d1`** and **`d2`** is a double 1.0. It has to be converted to an **`int`** 1 using the **`(int)`** operator. On line 11, the **`int`** values in **`i`** and **`j`** are converted to double using the **`(double)`** operator before being subtracted and assigned to the variable **`d3`**, which is of type **`double`**. On line 12, the two **`double`** values in **`i`** and **`j`** are operated first before the resulting **`int`** value is converted to **`double`**.

Automatic Type Conversion and Explicit Type Casting

Explicit casting must be used (i.e. cast operators must be used) when converting wider data types (i.e. data types with larger numbers of bytes) to narrower data types (i.e. data types with smaller numbers of bytes). However, converting narrower data types to wider data types can be done implicitly without using cast operators. In other words, Java can convert data types automatically if the value ranges of the destination can cover the original data types. For example, **`double d = 2.5f;`** can be done without any explicit use of a cast operator, while **`float g = 2.5;`** will result in a compilation error. To perform such an assignment, **`float g = (float) 2.5;`** must be used.

The following table shows the possibility of converting among primitive data types. The label 'A' means that the data type on that row can be automatically converted to the data type in that column. The label 'C' means that the associated cast operators are required. The label 'X' means that such a conversion is not allowed in Java.

from\to	byte	short	int	long	float	double	char	boolean
byte		A	A	A	A	A	C	X
short	C		A	A	A	A	C	X
int	C	C		A	A*	A	C	X
long	C	C	C		A*	A*	C	X
float	C	C	C	C		A	C	X
double	C	C	C	C	C		C	X
char	C	C	A	A	A	A		X
boolean	X	X	X	X	X	X	X	

* indicates that precision lost might occur from the conversion.

Table 8: Data type conversion table

Note that the conversion from a wider data type to a narrower data type is called *narrowing*, while the opposite is called *widening*.

Expressions with Multiple Data Types

It is not uncommon to run into an expression that contains operators whose operands are values and/or variables of different data types. In Java, the data type of the resulting value from evaluating any one expression can be determined precisely. Before we state the rule used for determining it, let's observe the expressions in the following example.

Example 19: Resulting data type of an expression

```
public class DataTypesFromExp                                    1
{                                                                2
```

(continued on next page)

(continued from previous page)

```
        public static void main(String[] args)                    3
        {                                                         4
            int k = 10;                                           5
            System.out.println("k/3          = "+k/3);            6
            System.out.println("k/(float)3   = "+k/((float)3));    7
            System.out.println("k/3.0        = "+k/3.0);          8
        }
}
```

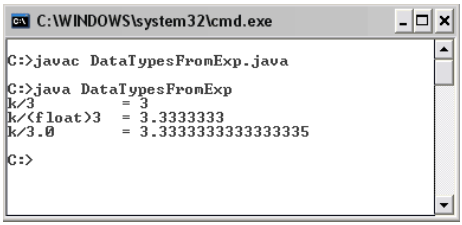


Figure 58: A Java program showing expressions with operands being of multiple data types

On line 6, the expression **`k/3`** is an operation between two **`int`** values. The result of this expression is a value of type **`int`**, as we can see no decimal points in the output. Doing simple division by hand, we can see that 10/3 results in 3.333… However, following some rules which we are going to discuss later, the resulting data type of **`k/3`** must be an **`int`**. Thus, 3.333… is truncated to the integer 3. Following the same set of rules, the resulting type of an operation between an **`int`** and a **`float`**, such as **`k/((float) 3)`**, is a **`float`** and the resulting type of an operation between an **`int`** and a **`double`**, such as **`k/3.0`**, is a **`double`**.

As promised, here is the rule that governs the resulting date types of expressions with operands with mixed data types.

> Consider **`A $ B`** where **`$`** is a binary operator, while **`A`** and **`B`** are its operands. When **`A`** and **`B`** are of the same data type, the resulting data type is that data type.

- If either operand is of type **`double`**, the other is converted to **`double`**.
- Otherwise, if either operand is of type **`float`**, the other is converted to **`float`**.
- Otherwise, if either operand is of type **`long`**, the other is converted to **`long`**.
- Otherwise, both operands are converted to type **`int`**.

The table below gives us some example statements and their results.

Statement	Value of **`x`**	Statement	Value of **`x`**
float x = 8/9;	0.0f	int x = (int)(1.5+1.5)	3
double x = 10d + 8/9	10.0	double x = (int)1.5+1.5	2.5
double x = (10.0+8)/9	2.0	double x = 3/4.0	0.75
float x = 8.0+9;	error	double x = 3/4*4/3	0.0
double x = (1/4)*(12d-4)	0.0d	double x = 4/3*(float)3/4	0.75

Table 9: Data types from expression evaluation

Limitation on Floating Point Computation

Consider the source code listed below. First, a variable **`d`** is defined and initialized to 0.0. The program adds 0.1 to **`d`** ten times. The program shows the current value of **`d`** after each time 0.1 is added to **`d`**.

```
public class LimitedPrecision1
{
        public static void main(String[] args)
        {
            double d = 0.0;
            double increment = 0.1;
            System.out.println("Original d \t\t="+d);
            d += increment;
            System.out.println("d + 1 increments \t="+d);
            d += increment;
            System.out.println("d + 2 increments \t="+d);
            d += increment;
            System.out.println("d + 3 increments \t="+d);
```

(continued on next page)

(continued from previous page)

```
            d += increment;
            System.out.println("d + 4 increments \t="+d);
            d += increment;
            System.out.println("d + 5 increments \t="+d);
            d += increment;
            System.out.println("d + 6 increments \t="+d);
            d += increment;
            System.out.println("d + 7 increments \t="+d);
            d += increment;
            System.out.println("d + 8 increments \t="+d);
            d += increment;
            System.out.println("d + 9 increments \t="+d);
            d += increment;
            System.out.print("d + 10 increments \t="+d);
        }
}
```

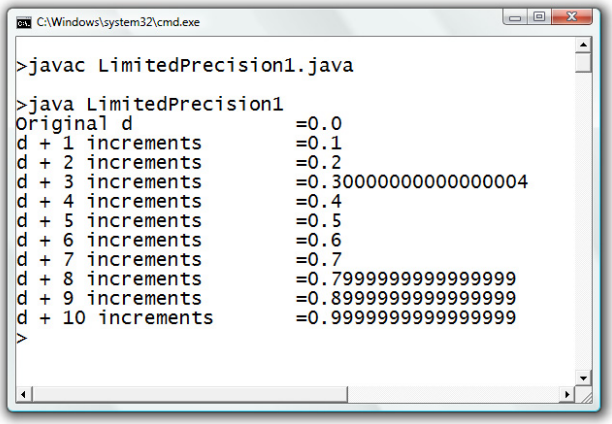


Figure 59: A demonstration of the fact that precisions are limited in the floating-point calculation using computers

The above figure shows the output of the program. With simple mathematics, one will definitely say that the eventual value of `d` is 1.0. However, when this program is executed, we have found that the eventual value of `d` is very close to but not exactly 1.0. This is because of the imprecision of representing floating point values with binary representation. Thus, some decimal points cannot retain their exact

decimal values when represented with binary representation. And, 0.1 is one of them.

An Issue on Floating Point Value Comparison

With the fact about limited precisions in the floating point calculation kept in mind, one should be aware that floating point values should not be compared directly with relational equality operators (== and !=). A good practice for comparing a floating point value A and a floating point value B is to compute $d = |A - B|$, and see whether d is small enough. If so, you could consider that A equals to B.

Overflow and Underflow

It is possible that the results of some arithmetic operations are larger or smaller than what numeric values can handle. When a value is larger than the biggest value that a data type can handle, it is said that an *overflow* has occurred.

Programmers should be careful about the range of value that a data type can handle. When an overflow of a value of the type **`int`** occurs, it neither leads to any compilation errors or warnings nor causes the program to terminate when executed. However, overflowed values could causes unexpected (but explainable) results.

For floating point values, Java has special **`float`** and **`double`** values representing a positive value that is larger than the largest positive value that the respective data type can handle as well as values representing a negative value that is smaller (more negative) than the smallest negative value that the data type can handle. The former is the value **`Infinity`**, and the latter is the value **`-Infinity`**.

If a floating point value is too close to zero for its associated data type can handle, that value might be rounded to 0. This situation is called *underflow*. Just like overflow, underflow might cause some arithmetic computations to yield unexpected results.

Example 20: Oops! They are too big.

The program Overflow1.java listed below shows an example of when overflows have occurred. A variable **`veryBigInteger`** of type **`int`** is declared and set to a big integer value of 2,147,483,647, which is still in the range that **`int`** can handle. Adding 1 to 2,147,483,647 would normally result in 2,147,483,648. However, such a value is too large for an **`int`**. Thus, the program gives an unexpected answer. A similar situation happens when 1 is subtracted from a very small number -2,147,483,648 and when 2,147,483,647 is multiplied by 2. Notice that the program does not warn or give any errors or exceptions when an overflow occurred with **`int`**. Thus, programmer should be aware of this situation and prevent it for happening.

```
public class Overflow1                                                  1
{                                                                       2
        public static void main(String[] args)                         3
        {                                                               4
            int veryBigInteger = 2147483647;                            5
            System.out.print("veryBigInteger\t\t= ");                   6
            System.out.println(veryBigInteger);                         7
            System.out.print("veryBigInteger+1\t= ");                   8
            System.out.println(veryBigInteger+1);                       9
            System.out.print("veryBigInteger*2\t= ");                   10
            System.out.println(veryBigInteger*2);                       11
            int verySmallInteger = -2147483648;                         12
            System.out.print("verySmallInteger\t= ");                   13
            System.out.println(verySmallInteger);                       14
            System.out.print("verySmallInteger-1\t= ");                 15
            System.out.println(verySmallInteger-1);                     16
        }                                                               17
}                                                                       18
```

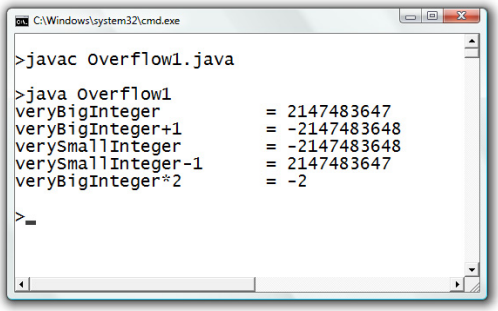


Figure 60: Demonstration of several occurrences of overflow for int

Example 21: Once an infinity, always an infinity

The program Overflow2.java shows an example of when an overflow occurred with **`double`**. However, Java has a floating point value (both **`float`** and **`double`**) that can handle a very large number. Such a value is **`Infinity`**. Remember that **`Infinity`** is a floating point value. So, it can be treated as one. **`-Infinity`** represents a floating point value that is very small. Also, notice the effect of **`Infinity`** from the result of line 10.

```
public class Overflow2                                                    1
{                                                                         2
        public static void main(String[] args)                           3
        {                                                                 4
            double d = 1.0e308;                                           5
            double dSquare = d*d;                                         6
            System.out.println("d     = "+d);                           7
            System.out.println("d^2   = "+dSquare);                      8
            System.out.println("-d^2  = "+(-dSquare));                   9
            System.out.println("sqrt(d^2) = "+Math.sqrt(dSquare));       10
        }                                                                 11
}                                                                         12
```

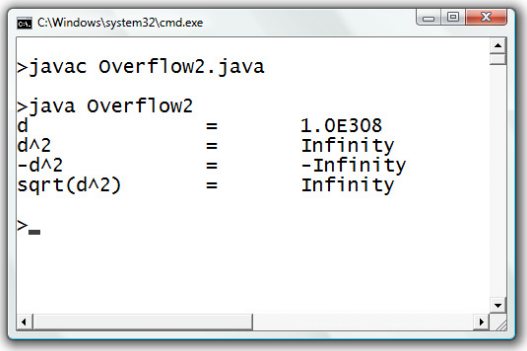


Figure 61: A Java program that results in the double-valued Infinity

Example 22: Too close to zero

The program Underflow1.java shows an example when a value becomes closer to zero than its associated data type can handle, it is rounded to zero. In this example, **`d`** is set to 1.0×10^{-323}, which is still in the precision

range of a **double**. Divide that value by 10 results in the value that is closer to zero than what **double** can handle. Consequently, the result of the expression **d/10*10** becomes zero since the division is executed first and it results in a zero. Notice the difference between the value of **p** and **g**.

```
public class Underflow1                                          1
{                                                                2
        public static void main(String[] args)                  3
        {                                                        4
            double d = 1.0e-323;                                 5
            double p = d/10*10;                                  6
            double g = 10*d/10;                                  7
            System.out.println("d        = "+d);                 8
            System.out.println("d/10*10  = "+p);                 9
            System.out.println("10*d/10  = "+g);                 10
        }                                                        11
}                                                                12
```

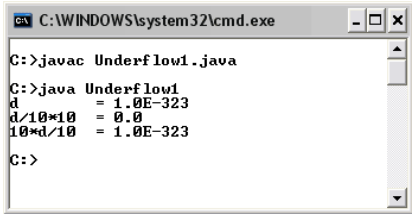


Figure 62: A demonstration of cases when underflow occurs

Numbers Divided by Zero

Mathematically when a number is divided by zero, it is sometimes considered as an undefined case or it can be thought of as resulting in a infinitely large number. In Java, handling of such a case depends on the data types of the values involved in the division operation. For **int** operations, the compiler will not catch such a case during the compilation and the program will produce a kind of error (technically called exception) producing at the execution of the program and

terminate the program at the step when the divided-by-zero event happens. For operations that result in **`float`** or **`double`** values, a number divided by zero will not cause the program to terminate and the resulting floating point values will be the **`Infinity`** or **`-Infinity`** values.

When a floating point value of zero is divided by another zero, the resulting value is **`NaN`**, which stands for "Not a Number". **`NaN`** is also used to represent a square root of negative number.

Example 23: Dividing by zeros: Types matter.

```
public class DivByIntZero1 {                                    1
    public static void main(String [] args){                   2
        int number = 4, zero = 0;                               3
        System.out.println(number/zero);                        4
    }                                                           5
}                                                               6
```

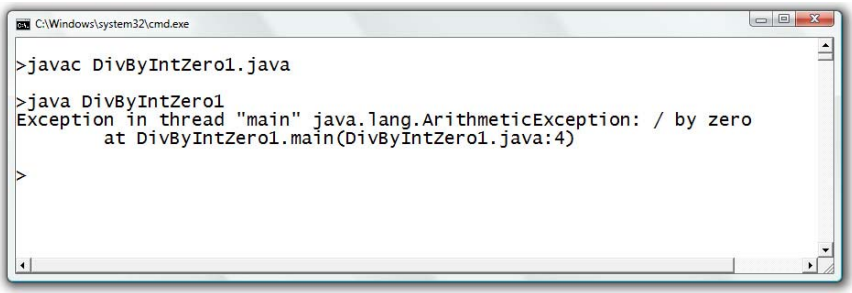


Figure 63: A Java program that produces the divided-by-zero exception

In this example, an **`int`** variable **`zero`** is intentionally assigned with the **`int`** value of 0 in order to demonstrate that when it is used as a denominator in the input of *System.out.println()* on line 4, the program produces an exception that causes the program to terminate. It is unsurprising that the compiler does not complain anything during the compilation since it does not check for values of all variables used to divide other values. Therefore, it cannot catch this during the time the program gets compiled.

Now let's look at another case in which floating-point numbers are divided by zeros.

```
public class DivByFloatingPointZero1 {                                  1
    public static void main(String [] args){                            2
        float f = 4.0f, floatZero = 0.0f;                               3
        double d = 4.0, doubleZero = 0.0;                               4
        System.out.println();                                           5
        System.out.print("f/floatZero\t=\t");                           6
        System.out.println(f/floatZero);                                7
        System.out.print("d/doubleZero\t=\t");                          8
        System.out.println(d/doubleZero);                               9
        System.out.print("-f/floatZero\t=\t");                          10
        System.out.println((-f)/floatZero);                             11
        System.out.print("-d/doubleZero\t=\t");                         12
        System.out.println((-d)/doubleZero);                            13
        System.out.print(" 0.0f/ 0.0f\t=\t");                           14
        System.out.println(floatZero/floatZero);                        15
        System.out.print(" 0.0 / 0.0\t=\t");                            16
        System.out.println(doubleZero/doubleZero);                      17
        System.out.print("-0.0f/ 0.0f\t=\t");                           18
        System.out.println((-floatZero)/floatZero);                     19

        System.out.print("-0.0 / 0.0\t=\t");                            20
        System.out.println((-doubleZero)/doubleZero);                   21
    }                                                                   22
}                                                                       23
```

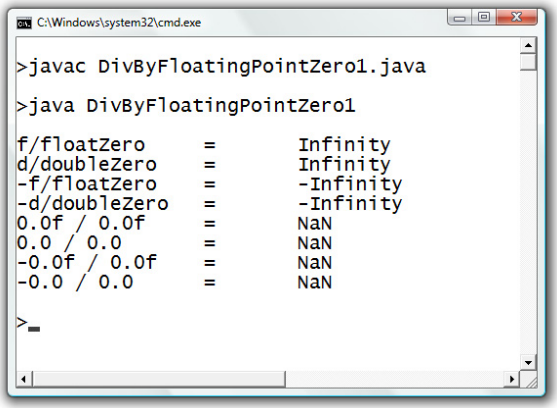


Figure 64: A Java program with expressions evaluated to the double-valued Infinity, -Infinity and NaN

Observe the result of this program, we can see that the program prints out **`Infinity`** or **`-Infinity`** in the cases that floating point numbers are divided by zero. Both values differ in the way that the former one is positive and the latter one is negative. Also, we can observe the occurrences of **`NaN`** due to the fact that zeros are divided by zeros.

Example 24: An 'infinity' has its data type

The following source code shows errors that emphasize the fact that **`Infinity`** is a value that is subjected to strong data typing as well as other values.

```
public class TypeOfInfinity {                                    1
    public static void main(String [] args){                    2
        float f1 = 4.0/0.0;                                      3
        float f2 = 0.0/0.0;                                      4
        float f3 = 4.0f/0.0f;                                    5
        float f4 = 0.0f/0.0f;                                    6
    }                                                            7
}                                                                8
```

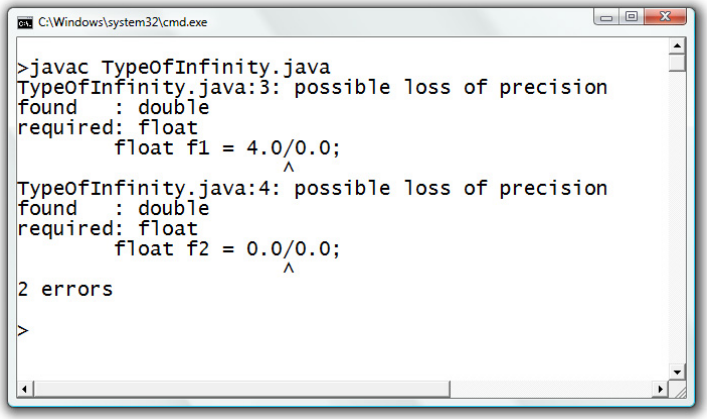


Figure 65: Compilation errors due to data type mismatch in the cases when numbers are divided by zeros

From the screenshot showing the errors, we can see that the Java compiler report two errors caused by the assignment statements on line 3 and on line 4. Although the expression **`4.0/0.0`** should be evaluated to

`Infinity` without any problems, its data type is double which cannot be directly assigned to a `float` variable. On line 4, the expression `0.0/0.0` results in the value `NaN` of the type double which also caused another error when assigned to a `float` variable. However, errors do not occur for the statements on line 5 and on line 6 since the resulting `Infinity` and `NaN` are of the type `float`.

Compound Assignment

One common task frequently found in general computer programming is when an operator is applied to the value of a variable, and then the result of the operation is assigned back to that variable. For example, `k = k + 7;`, `m = m*3;`, and `s = s+"ful";`. These can be written in a shorthanded fashion in Java.

> Let ◊ be a Java operator, `k` be a variable, and `m` be any valid Java expression. The operation:
>
> `k = k ◊ m;` can be written as `k ◊= m;`

For example,

`k = k + 1;` can be written as `k += 1;`
`i = i * 8;` can be written as `i *= 8;`
`j = j / 3;` can be written as `j /= 3;`
`s1 = s1 + s2;` can be written as `s1 += s2;`

Increment and Decrement

Java has special operators for incrementing and decrementing a numeric variable. The increment operator (++) add to the value of the variable to which it is applied. The decrement operator (--) subtract from the value of the variable to which it is applied.

`++k` and `k++` is equivalent to `k = k +1;` or, similarly, `k += 1;`

`--k` and `k--` is equivalent to `k = k – 1;` or, similarly, `k -= 1;`

The difference between when the increment or decrement operator is applied prior to the variable (prefix version) and after the variable (postfix version) becomes apparent when it is a part of a longer expression when the value of the variable is used. In such a situation, if an increment or decrement operator is applied prior to the variable, the value of that variable increases or decreases by one prior to being used as a part of the expression. In contrary, if it is applied after the variable, its value is used prior to the incrementing or the decrementing.

Consider the following example.

Example 25: Prefix-postfix frenzy

Observe the difference between the results of the prefix and the postfix versions of the increment operator used in the following program.

```
public class IncDecOp {                                          1
    public static void main(String [] args){                    2
        int x=1, y=1;                                            3
        System.out.println();                                    4
        System.out.println("Start at x = 1");                    5
        System.out.println("Line 6  ->" + x++);                  6
        System.out.println("Line 7  ->" + x);                    7
        System.out.println();                                    8
        System.out.println("Start at y = 1");                    9
        System.out.println("Line 10 ->" + ++y);                  10
        System.out.println("Line 11 ->" + y);                    11
    }                                                            12
}                                                                13
```

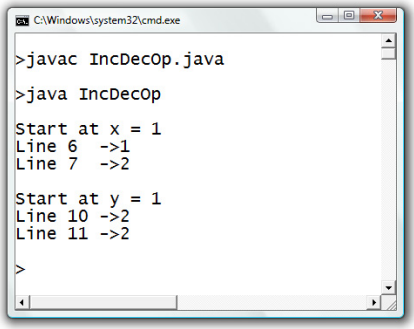


Figure 66: A Java program demonstrating the difference between the prefix and the postfix versions of the increment operator

Let's focus on the output due to the statement on line 6. In this statement, **`x++`** is used as a part of the expression supplied as the input to *System.out.println()*. Since the increment operator used here is the postfix version, the value of **`x`** (which is 1) is used as an operand for evaluating the expression input to the method prior to the increment of **`x`**. As we print the value of x out after that, we can observe that the resulting value of **`x`** is indeed incremented from 1 to 2, as the result of the increment operator.

In contrary to the postfix version, the increment expression in the statement on line 10 uses the prefix version on **`y`**. This makes the expression increase value of y by 1 before the increased value is used for the rest of the statement. Therefore, the value 2 is concatenated with **`"Line 10 ->"`** and printed out on screen.

Exercise

1. Specify the data type of these values.

```
9.0       8          15d          900F         258234
'8'       "888"      "16.0d"      15L          "8"
0x99      (int)9.1   1e1          256f         900L
1.0e10
```

2. Specify the data type of the values resulting from these operations.

```
1/2          9F+3D        (int)(5+5.0)
6.0%9        1.0*1/1      9+(double)4
1.5f+3       (int)5.0+5.0 (double)5+"6"
```

3. Explain *widening* and *narrowing* in the context of Java primitive date type conversion.

4. Explain why the following code segment causes a compilation error.

```
int x1,x2;
double y = 1.0;
x1 = (int)y;
x2 = 1L;
```

5. Determine the resulting value of the variable x in the following code segment.

```
double x;
int y = 90;
x = y/100;
System.out.println("x="+x);
```

6. Write a Java program that performs the following steps. Perform appropriate type casting when needed.

    a. Declare a **`float`** variable called **`f`**.
    b. Declare an **`int`** variable called **`k`**.
    c. Assign 22.5 to **`f`**.
    d. Assign the value of **`f`** to **`k`**.
    e. Convert the current value of **`f`** to short and print it out on screen.

7. Write a Java program that performs the following steps:

    a. Declare an **`int`** variable called **`k`**.
    b. Declare a **`double`** variable called **`pi`** and initialize it to the value of π
    c. Calculate the smallest integer that is bigger than the square of the value in **`pi`**, and assign the resulting value to **`k`**. (Do not forget type casting.)

8. Determine what will be shown on screen if the following statements are executed.

```
float f = 500F, g = 500F;
System.out.println(f/0);
System.out.println(-f/0);
System.out.println((f-500)/(f-g));
System.out.println(-(f-500)/(f-g));
```

9. Determine the value of x when the following code segment is executed.

```
double x = 1;
x += 3;
x *= 10;
x -= 10;
x /= 5;
x %= 5;
```

10. In one valid Java statement, assign y with twice the value of **x** and then increase **x** by one. Assume that **x** and **y** are variables of type **int** which are correctly declared and initialized.

11. In one valid Java statement, make the value of **x** twice as big as its current value and then assign the value to **y**. Assume that **x** and **y** are variables of type **int** which are correctly declared and initialized.

12. Explain why the following code segment causes a compilation error.

```
int x;
(x++)++;
```

13. Determine what will be shown on screen if the following statements are executed.

```
int x=1,y=5;
System.out.print(++x+",,+y++);
```

14. Determine what will be shown on screen if the following statements are executed.

```
int x=1;
System.out.print(x+(x++)+(x++));
```

15. Determine what will be shown on screen if the following statements are executed.

```
int x=1;
System.out.print(x+(++x)+(++x));
```

16. Determine what will be shown on screen if the following statements are executed.

```
int x=1,y;
System.out.print(y=x++ +x);
```

Chapter 5: Using Objects

Objectives

Readers should

- Understand classes and objects.
- Be able to use class methods and data from existing classes.
- Be familiar with the *String* class and be able to use its methods.
- Be able to use the *BufferedReader* class as well as the *Scanner* class to get users' input from keyboards.
- Be able to process keyboard input as *String* and numeric values.

Classes and Objects

In real world, the word *class* is used to identify category of things, while *objects* of a class are instances of things belonging to that category.

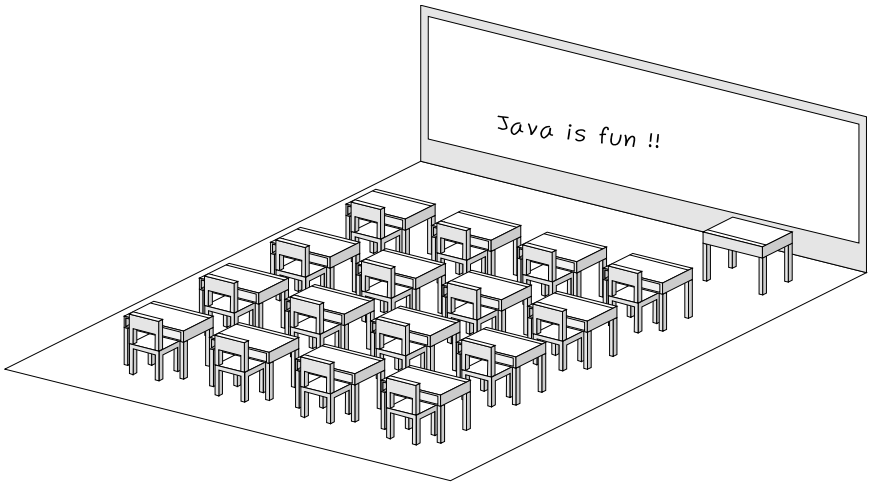


Figure 67: A classroom with 17 desks and 16 chairs

If we look at Figure 67 and say “There are 17 desks and 16 chairs in the classroom”, we could consider that each of the word ‘desk’ and the word ‘chair’ is a class. Each of the 17 desks is an instance or object of the class ‘desk’ and each of the 16 chairs is an instance or object whose type is different from each of the 17 desks. Objects of the same class must share some aspects of their properties. From the setting of the classroom in the figure, we might say that from the 17 desks, there are 16 desks that are student desks and the other one is a teacher desk. An instance of a student desk might have some properties that are absent from a teacher desk. At the same time, it might have some properties that are irrelevant to a teacher desk. However, both the student desk and the teacher desk share some common aspects. They are both desks. This means it is natural that they belong to the same class.

In Java, classes are data types which can be comparable to categories of things in real world. Classes are considered non-native data types. New classes can be created while primitive data types cannot. As in the real world, an object is an instance of a class. The data type of an object is the class it belongs to.

Consider the following Java statements. Note that, as we have mentioned, *String* is a class in Java.

```
String s1;
s1 = "Chocolate Chip";
```

In the first statement, a variable named **`s1`** is declared as a variable that is used for storing an object of the class *String*. In the second statement, an object of class **`String`** is created with the content "Chocolate Chip" and assigned to the variable **`s1`**. In other words, **`s1`** is made to point to the *String* object "Chocolate Chip". What have occurred can be depicted in Figure 68.

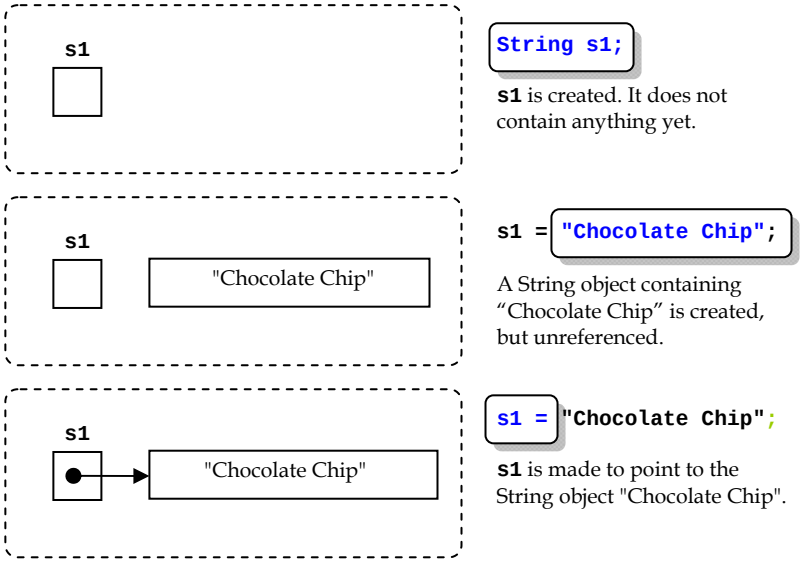


Figure 68: Declaration and assignment of a String object

Using Data and Methods provided in Classes

An object of a class contains data and methods. For example, there is a Java class names *Rectangle*, which is a data type used for representing a rectangle. The data contained in each object of this *Rectangle* class are **`height`**, **`width`**, **`x`**, and **`y`**, which stores necessary attributes that define a rectangle. Apart from data, the class also provides several methods related to using the rectangle, such as *getHeight()*, *getWidth()*, *getX()*, *getY()*, and *setLocation()*. This class might be illustrated in Figure 69. Note that the illustration used for describing class details here mainly adopts an industry-standard notation for describing classes but it differs in details for the sake of simplicity.

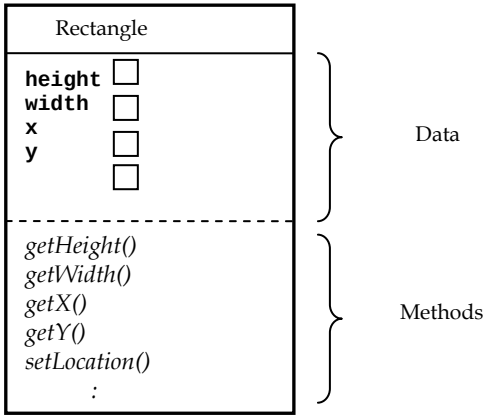


Figure 69: An abstract representation of the details of a class

Generally, when we write computer programs in Java, we make use of existing methods and data that have already been defined or made in some existing classes. The dot operator (.) is used for accessing data or methods from a class or an object of a class. We have already seen (and used) two methods that print message onto the screen since earlier chapters. Here, we discuss the meaning of them. Consider the two methods below.

```
System.out.print("Strawberry Sundae");
System.out.println("Banana Split");
```

The four periods seen in both statements above are the dot operator. *System* is a class in a standard Java package. This class contains an object called **`out`**, whose class is a class called *PrintStream*. Thus, using the dot operator, we refer to this **`out`** object in the class *System* by using **`System.out`**. Consequently, the *PrintStream* class contains *print()* and *println()*, and we can access the two methods using **`System.out.print()`** and **`System.out.println()`**.

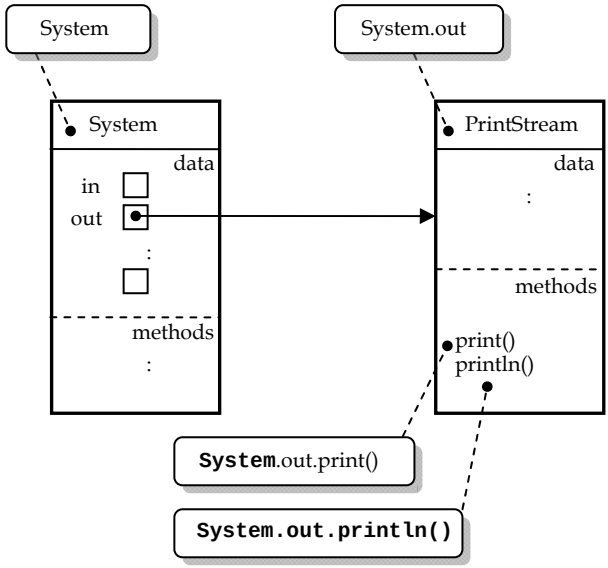


Figure 70: An abstract representation classes, objects, and methods involving in System.out.print() and System.out.println()

Some data and methods can be accessed by using the dot operator with the name of the class while some can be accessed by using the name of the variable storing the object of that class. Data and methods that are accessed via the class name are called *class* (or *static*) *data* and *class* (or *static*) *methods*. Data and methods that are accessed via the object name are called *instance* (or *non-static*) *data* and *instance* (or *non-static*) *methods*. At this point, you are not expected to know that whether the data and methods that you have never come across before are associated with classes or instances (objects). Just make sure you understand what you are doing when accessing ones.

Example 26: Area of a circle

```
public class AreaOfCircle                                          1
{                                                                  2
        public static void main(String[] args)                    3
        {                                                          4
            double area, r = 10;                                   5
```

(continued on next page)

(continued from previous page)

```
            String s1 = "The Area of a circle with ";          6
            String s2 = " r = ";                               7
            String s3 = " is ";                                8
            String s4;                                         9
            area = Math.PI*Math.pow(r,2);                      10
            s4 = s1.concat(s2);                                11
            System.out.println(s4+area);                       12
        }                                                      13
}                                                              14
```

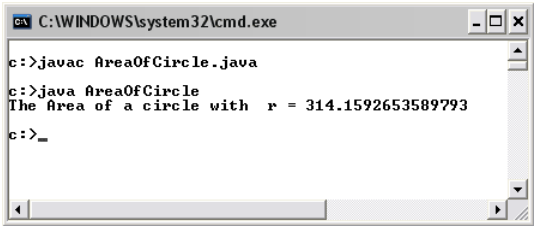


Figure 71: A program calculating the area of a circle

This Java program calculates the area of a circle with radius **`r`**, where **`r`** equals 10. On line 10, we calculate the area by multiplying **`Math.PI`** with **`Math.pow(r,2)`**. The former expression refers to the π value that is defined in a constant value names **`PI`** in the *Math* class. The later is the activation of a method called *pow()* that is also defined in the *Math* class. **`pow(r,2)`** computes the square of **`r`**. Notice that we do not need to create an object of the *Math* class but we access the data and method from the name of the class directly.

On line 11, we make use of a method called *concat()*. It is accessed from a variable that contains a *String* object. **`s1.concat(s2)`** returns a *String* object resulting from the concatenation of the *String* object in s1 and the *String* object in **`s2`**. Also, on line 11, the concatenated *String* object is assigned to **`s4`**.

Useful String methods

Let's look at some methods that we can use from a *String* object. The methods discussed here do not make the complete list of the methods provided by *String*. Examples are given so that you can see what each method does as well as practice your code reading skill at the same time.

charAt()

Let **`s`** be a *String* object and **`i`** be an **`int`**. **`s.charAt(i)`** returns the **`char`** value at the **`i`** th index.

length()

Let **`s`** be a *String* object. **`s.length()`** returns the **`int`** value equals to the length of the *String*.

Example 27: Demonstration of String methods (1)

Consider the following Java program.

```
public class CharAtDemo                                                 1
{                                                                       2
       public static void main(String[] args)                          3
       {                                                                4
           String s = "ABCD\nEFGH";                                     5
           char c;                                                      6
           System.out.println("s = ");                                  7
           System.out.println(s);                                       8
           c = s.charAt(0);                                             9
           System.out.println("charAt(0)="+c);                          10
           c = s.charAt(1);                                            11
           System.out.println("charAt(1)="+c);                         12
           c = s.charAt(5);                                            13
           System.out.println("charAt(5)="+c);                         14
           System.out.print("The length of this string is ")           15
           System.out.println(s.length()+" characters");               16
           c = s.charAt(s.length()-1);                                 17
           System.out.println("The last char ="+c);                    18
       }                                                               19
}                                                                      20
```

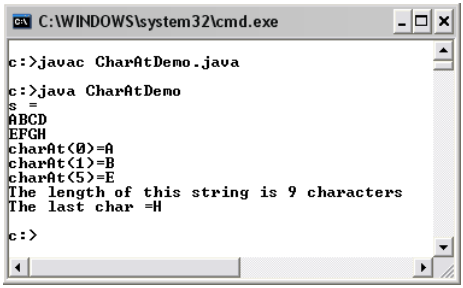


Figure 72: Demonstration of using charAt() and length()

From the above Java program, the *String* **s** contains 9 characters, which are **'A', 'B', 'C', 'D', '\n', 'E', 'F', 'G',** and **'H'**. Notice that an escape sequence is considered a single character. On line 9, line 11, and line 13, the characters at 0, 1, and 5 which are **'A', 'B'** and **'E'** are assigned to the **char** variable **c**. Then, **c** is printed out to the screen after each assignment. On line 16, and line 17, the length of the *String* in **s** is extracted via the method *length()*. Be aware that, the first index of a **String** is 0, so the location of the last character is **s.length()-1**.

---

concat()

Let **s** be a *String* object and **r** be another *String* object. **s.concat(r)** returns an new *String* object whose content is the concatenation of the *String* in **s** and **r**.

Example 28: Demonstration of String methods (2)

```
public class ConcatDemo                                          1
{                                                                2
       public static void main(String[] args)                   3
       {                                                         4
           String s1 = "First";                                  5
           String s2 = "Second";                                 6
           String s3, s4;                                        7
                                                                 8
           s3 = s1.concat(s2);                                   9
           s4 = s2.concat(s1);                                   10
           System.out.println("s1 is "+s1);                      11
```

(continued on next page)

(continued from previous page)

```
            System.out.println("s2 is "+s2);                          12
            System.out.println("s3 is "+s3);                          13
            System.out.println("s4 is "+s4);                          14
                                                                      15
            String s5 = "AB".concat("CD").concat("EF");               16
            System.out.println("s5 is "+s5);                          17
        }                                                             18
}                                                                     19
```

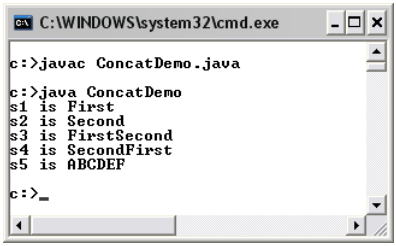


Figure 73: Demonstration of using concat()

Notice the difference between **`s1.concat(s2)`** and **`s2.concat(s1)`**. Also note that invoking the method *concat()* from a *String* **`s`** creates a new *String* object based on **`s`** and the *String* input into the parentheses, it does not change the value of the original *String* object. On line 16, we show two things. Firstly, we can invoke *String* methods directly from a *String* object without having to be referred to by a variable, i.e. **`"AB".concat("CD")`** can be done without any errors. Secondly, since **`"AB".concat("CD")`** results in a new *String* object, we can call a *String* method from it directly, e.g. **`"AB".concat("CD").concat("EF")`**, and the result is **`"ABCDEF"`**, as expected.

---

indexOf()

Let **`s`** be a *String* object and **`c`** be a **`char`** value. **`s.indexOf(c)`** returns the index of the first **`c`** appearing in the *String*. It returns -1 if there is no **`c`** in the *String*. If **`i`** is an **`int`** value equals to the Unicode value of **`c`**, **`s.indexOf(i)`** returns the same result. A *String* **`r`** can also be used in the place of **`c`**. In that case, the method finds that *String* inside the

String **s**. If there is one, it returns the index of the first character of **r** found in the *String* **s**. Again, it returns -1 if **r** is not found in **s**.

lastIndexOf()

lastIndexOf() works similarly *to indexOf()* but it returns the index of the last occurrence of the input character or the index of the first character in the rightmost occurrence of the input *String*.

Example 29: Demonstration of String methods (3)

The following program demonstrate some results of using *indexOf()* on a *String* object.

```
public class IndexOfDemo                                                    1
{                                                                           2
    public static void main(String[] args)                                 3
    {                                                                       4
      String s = "oxx-xo--xoXo";                                            5
      System.out.println("The first 'x' is at "+s.indexOf('x'));            6
      System.out.println("The first 'o' is at "+s.indexOf('o'));            7
      System.out.println("The first '-' is at "+s.indexOf(45));             8
      System.out.println("The first 'X' is at "+s.indexOf('X'));            9
    }                                                                       10
}                                                                           11
```

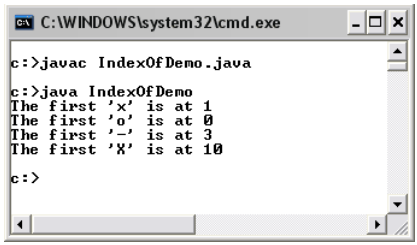


Figure 74: Demonstration of using indexOf()

From the program, we can see that the first occurrence of the character 'x' is at position 1 and it is at position 0 for 'o'. You should be reminded that the first position is indexed as the position 0. These can be compared with the result of the statements on line 6 and line 7.

The statement on line 8 shows an example of when the input to *indexOf()* is an **int** value. Since the Unicode of **'-'** is 45, the method returns 3 as it is the first position that **'-'** occurs.

Also, make sure you remember that Java is a case-sensitive language. That is why the result of **s.indexOf('x')** is different than the one of **s.indexOf('X')**.

The following program use *String* objects as inputs to *indexOf()*.

```
public class IndexOfDemo2                                          1
{                                                                  2
       public static void main(String[] args)                     3
       {                                                           4
           String s = "Chulalongkorn University";                  5
           System.out.println(s);                                  6
           System.out.println("Univ is at "+s.indexOf("Univ"));     7
           System.out.println("0123 is at "+s.indexOf("0123"));     8
       }                                                           9
}                                                                  10
```

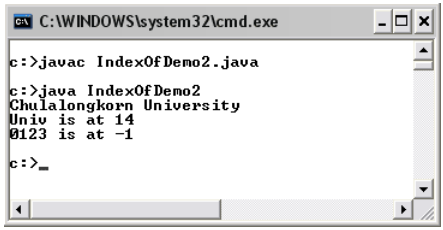


Figure 75: Demonstration of using lastIndexOf()

The *indexOf()* methods on line 7 and line 8 find the first occurrences of the input *String* objects in **s**. Each of them returns the position of the first character of the first occurrence of the *String* supplied as input. Figure 76 illustrates the position of "Univ" in **s**.

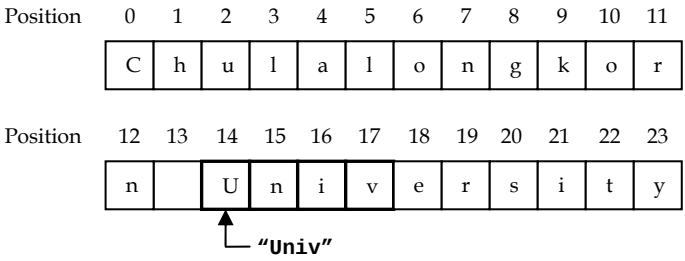


Figure 76: Finding Strings in another String

The int value -1 is returned when the string cannot be found, as we can see from the result of the statement on line 8.

Example 30: Demonstration of String methods (4)

This example demonstrate the use of *lastIndexOf()*.

```
public class IndexOfDemo3                                          1
{                                                                  2
  public static void main(String[] args)                          3
  {                                                                4
    String s = "say ABC ABC ABC";                                  5
    System.out.println(s);                                         6
    System.out.print("lastIndexOf(\'B\') =");                      7
    System.out.println(s.lastIndexOf('B'));                        8
    System.out.print("lastIndexOf(\"AB\")=");                      9
    System.out.println(s.lastIndexOf("AB"));                       10
  }                                                                11
}                                                                  12
```

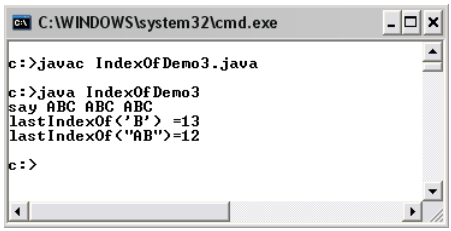


Figure 77: Demonstration of using lastIndexOf()

On line 8, the method is input with a char value while it is input with a *String* object on line 10. In the case of *String*, the method returns the position of the first character of the last occurrence.

---

startsWith()

Let **s** be a *String* object and **prefix** be another *String* object. **s.startsWith(prefix)** returns **true** if the *String* **s** starts with **prefix**. Otherwise, it returns **false**.

endsWith()

Let **s** be a *String* object and **suffix** be another *String* object. **s.endsWith(suffix)** returns **true** if the *String* **s** ends with **suffix**. Otherwise, it returns **false**.

trim()

Let **s** be a *String* object. **s.trim()** returns a new *String* object, which is a copy of **s**, but with leading and trailing whitespaces omitted.

Example 31: Demonstration of String methods (5)

```
public class TrimDemo                                                  1
{                                                                      2
  public static void main(String[] args)                               3
  {                                                                    4
    String s1 = " Computer Engineering   ";                            5
    String prefix = "Computer";                                        6
    String suffix = "ing";                                             7
    System.out.print("\""+s1+"\" has \""+prefix);                      8
    System.out.print("\" as a prefix:\t");                            9
    System.out.println(s1.startsWith(prefix)+".");                     10
    System.out.print("\""+s1+"\" has \""+suffix);                      11
    System.out.print("\" as a suffix:\t");                            12
    System.out.println(s1.endsWith(suffix)+".");                       13
                                                                       14
    String s2 = s1.trim();                                             15
    System.out.print("\""+s2+"\" has \""+prefix);                      16
    System.out.print("\" as a prefix:\t");                            17
    System.out.println(s2.startsWith(prefix)+".");                     18
    System.out.print("\""+s2+"\" has \""+suffix);                      19
    System.out.print("\" as a suffix:\t\t");                          20
```

(continued on next page)

```
(continued from previous page)
    System.out.println(s2.endsWith(suffix)+".");                    21
  }                                                                 22
}                                                                   23
```

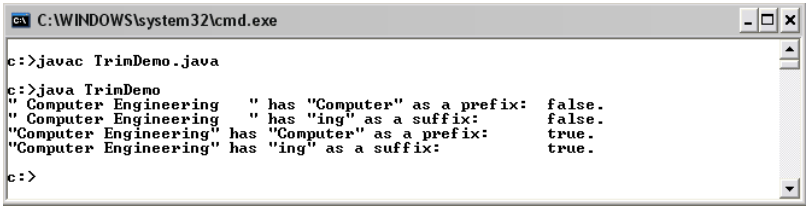


Figure 78: Demonstration of using trim(), startsWith(), and endsWith()

The String **s1** contains one whitespace character at the beginning and three of them at the end. A new *String* object is created with these leading and trailing whitespace characters trimmed before being assigned to **s2**. Since **s2** contains **"Computer"** right at the beginning and **"ing"** right at the end of the *String*, **s2.startsWith(prefix)** and **s2.endsWith(suffix)** return the **boolean** value **true**.

---

substring()

Let **s** be a *String* object. **s.substing(a,b)**, where **a** and **b** are **int** values, returns a new *String* object whose content are the characters of the *String* **s** from the **a**th index to the **(b-1)**th index. If **b** is omitted the substring runs from **a** to the end of **s**.

toLowerCase()

Let **s** be a *String* object. **s.toLowerCase()** returns a new *String* object which is a copy of **s** but with all uppercase characters converted to lowercase.

toUpperCase()

Let **s** be a *String* object. **s.toUpperCase()**returns a new *String* object which is a copy of **s** but with all lowercase characters converted to uppercase.

Example 32: Demonstration of String methods (6)

Sometimes we might want to extract portion of a String object in which we know that a certain number of sub-strings reside as well as that a certain character is used as the delimiter. In this example, we show a program that extracts sub-strings from "One:Two:Three" when the character ':' is thought of the delimiter that divides the three words: "One", "Two", and "Three".

```
public class SubStringDemo1 {                                          1
    public static void main(String [] args){                          2
        String s = "One:Two:Three", s1,s2,s3;                          3
        s1 = s.substring(0,s.indexOf(':'));                            4
        s2 = s.substring(s.indexOf(':')+1,s.lastIndexOf(':'));         5
        s3 = s.substring(s.lastIndexOf(':')+1);                        6
        System.out.println(s1);                                        7
        System.out.println(s2);                                        8
        System.out.println(s3);                                        9
    }                                                                  10
}                                                                      11
```

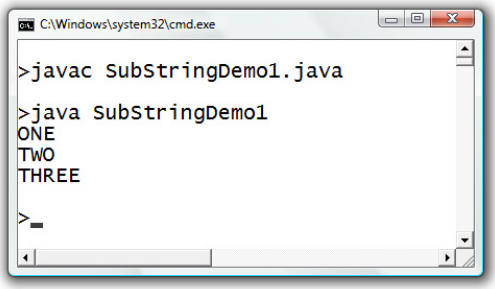


Figure 79: Demonstration of using substring() with methods that find the position of characters in String objects as well as toUppercase()

The word "One" is extracted by creating a new *String* object from `s` starting from the position 0 upto just before the first occurrence of ':' The word "Two" is from the position after the first occurrence of ':' upto just before the position of the last (second, in this case) occurrence of ':'. The word "Three" is from the position after the last occurrence of ':' up until the end of `s`.

In this example, we also convert every characters to their uppercases using *toUppercase()* before printing them on to the screen.

---

valueOf()

valueOf() is a static or class method provided by the *String* class. It creates a new *String* object whose value is the corresponding *String* representation of the value input to the method. Recall that to use a class method, we use the dot operator with the name of the class.

Reading Input String from Keyboards

It is usually a common requirement to obtain values from the user of the program via keyboards. In Java, this capability is provided by some methods, already defined in classes. A class called *BufferedReader* provides a method that read characters from keyboard input, until a newline character is found, and store the characters into a *String* object. This method is called *readLine()*. Note that the newline character (\n) signaling the end of the input is not included in the *String*.

First, since we are going to use the *BufferedReader* class, which is not packages which are included by default, we need to let the compiler know where to look for the definition of this class by adding the following statement in to our source code on a line prior to the start of our program's definition.

```
import java.io.*;
```

Then, we need to create an object of class *BufferedReader* by using the following statement.

```
BufferedReader stdin = new BufferedReader(new
InputStreamReader(System.in));
```

This statement creates a variable named **`stdin`** that refers to a *BufferedReader* object. For simplicity, we will say that **`stdin`** is a *BufferedReader* object. It is perfectly fine that you use exactly this

statement to create a *BufferedReader* object. Detailed explanation is omitted here.

Once a *BufferedReader* object is created, we can access the *readLine()* method from that object. For example, we can use the following statement to read keyboard input to a *String* object called **s**. Note that **stdin** is the object we created in the previous statement.

```
String s = stdin.readLine();
```

Once the statement is executed, the program waits for the user to type in the input until a newline character is entered. This input can be used later in the program from the *String* **s**.

Example 33: Greeting the users by their names

The following program asks the user to input his/her first and last name. Then it prints a message containing the names on to the screen. Notice that another thing that is required to be added is **throws IOException** in the header of the *main()* method. Again, explanation is omitted until you learn about *exceptions* in Java. At this time, make sure you do not forget to adds it in your program when *readLine()* is used in the *main()* method.

```
import java.io.*;                                                    1
public class Greeting                                                2
{                                                                    3
       public static void main(String[] args) throws IOException    4
       {                                                             5
           String firstname, lastname;                               6
           BufferedReader stdin =                                    7
               new BufferedReader(                                   8
                   new InputStreamReader(System.in));                9
           System.out.print("Please enter your firstname:");        10
           firstname = stdin.readLine();                             11
           System.out.print("Please enter your lastname:");        12
           lastname = stdin.readLine();                              13
           System.out.println("----------------------------");      14
           System.out.println("Hello "+firstname+" "+lastname);     15
           System.out.println("----------------------------");      16
       }                                                             17
}                                                                    18
```

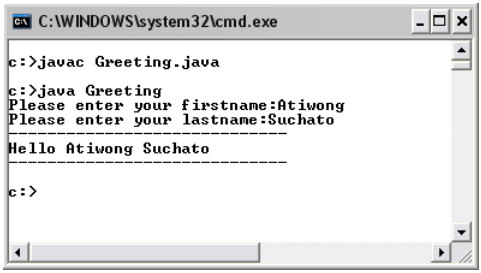


Figure 80: A program that reads two String inputs from the keyboard

In this example, italicized expressions are what you need to pay attention to. On line 1, the **`import`** statement tells the compiler about a location that it should look if there appear to be non-standard methods. In the statement that spans line 7 to line 9, a *BufferedReader* object, which we name it **`stdin`**, is created using the statement mentioned earlier. On line 11 and line 13, the method *readLine()* is used to bring in the keyboard inputs. It is a common practice that messages are shown prior to the execution of *readLine()* in order to instruct users about what they should be doing. Such messages are shown using *print()* on line 10 and line 12.

Converting Strings to numbers

Since the *readLine()* method returns a *String* object and sometimes we expect the keyboard input to be numeric data so that we can process numerically, we need a way to convert a *String* object to an appropriate numeric value. Luckily, Java has provided methods responsible for doing so.

parseInt()

parseInt() is a static method that takes in a *String* object and returns an **`int`** whose value associates with the content of that *String*. *parseInt()* is defined in a class called *Integer*. Thus, we should know by now that calling a static method named *parseInt()* from the *Integer*

class takes the form: **`Integer.parseInt(s)`**, where **`s`** is a *String* object whose content we wish to convert to **`int`**.

parseDouble()

parseDouble() is a static method that takes in a *String* object and returns an **`double`** whose value associates with the content of that *String*. *parseDouble()* is defined in a class called *Double*. Again, calling *parseDouble()* takes the form: **`Double.parseDouble(s)`**, where **`s`** is a *String* object whose content we wish to convert to **`double`**.

Useful Methods and Values in Class Integer and Class Double

It is necessary to know that *Integer* is a class, not the primitive type **`int`**, and *Double* is another class, not the primitive type **`double`**. Furthermore, it might come in handy if you know some of the constants and static methods provided in these two classes (Apart from *parseInt()* and *parseDouble()*, of course).

Here are some of them.

`Integer.MAX_VALUE`
is an **`int`** holding the maximum value an **`int`** can have (2^{31}-1).

`Integer.MIN_VALUE`
is an **`int`** holding the minimum value an **`int`** can have (-2^{31}).

`Integer.toBinaryString(<an int>)`
returns a *String* of the **`int`** argument as an unsigned integer in base 2.

`Integer.toOctalString(<an int>)`
returns a *String* of the **`int`** argument as an unsigned integer in base 8.

`Integer.toHexString(<an int>)`
returns a *String* of the **`int`** argument as an unsigned integer in base 16.

Integer.toString(<an int>)
: returns the *String* representation of the **int** argument.

Double.MAX_VALUE
: is the largest positive finite value of type **double**.

Double.MIN_VALUE
: is the smallest positive nonzero value of type **double**.

Double.NaN
: is a Not-a-Number (**NaN**) value of type **double**.

Double.POSITIVE_INFINITY
: is the positive infinite value of type **double**.

Double.NEGATIVE_INFINITY
: is the negative infinite value of type **double**.

Double.isInfinite(<a double>)
: returns **true** if the **double** argument is infinitely large in magnitude.

Double.isNaN(<a double>)
: returns **true** if the **double** argument is an **NaN** value.

Double.toHexString(<a double>)
: returns the hexadecimal *String* of the **double** argument.

Double.toString(<a double>)
: returns the *String* representation of the **double** argument.

Example 34: Showing the binary representation of the input number

The program ShowBinary.java shown below is used for showing the binary representation of an integer input by the user. This program could act as a simple tool that helps you convert integers to its base 2 format. Make sure you go through the program and try to understand all of the statements.

```
import java.io.*;                                                    1
public class ShowBinary                                              2
```

(continued on next page)

(continued from previous page)

```
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
    String readStr;                                                  6
    int i;                                                           7
    BufferedReader stdin =                                           8
           new BufferedReader(new                                    9
                 InputStreamReader(System.in));                      10
    System.out.print("Enter an integer:");                         11
    readStr = stdin.readLine();                                      12
    i = Integer.parseInt(readStr);                                   13
    System.out.println("Binary -> "+Integer.toBinaryString(i));     14
  }                                                                  15
}                                                                    16
```

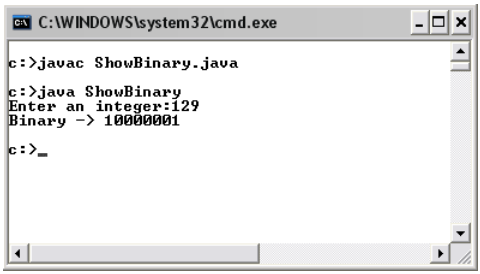


Figure 81: A program that reads two String inputs from the keyboard

The key idea to this program is that whenever you need to use the input entered from the keyboard via *readLine()* as numeric values, the input *String* object usually has to be converted to values in some numeric data types first. On line 13, we use *Integer.partInt()* to convert the input String object obtained from *readLine()* to **`int`** before using it as a numeric value further in the program.

Example 35: Selective substrings

This example shows a program that receives multiple keyboard inputs. Some are treated as strings, while some are converted to numbers.

```
import java.io.*;                                                    1
public class SelectiveSubString {                                    2
```

(continued on next page)

(continued from previous page)

```
  public static void main(String [] args) throws IOException      3
  {                                                               4
      String input,output;                                        5
      int start,len;                                              6
      BufferedReader in;                                          7
      in = new BufferedReader(                                    8
                new InputStreamReader(System.in));                9
      System.out.print("Enter a message:");                       10
      input = in.readLine();                                      11
      System.out.print("Enter the starting position:");           12
      start = Integer.parseInt(in.readLine());                    13
      System.out.print("Enter the length of the sub-string:");    14
      len = Integer.parseInt(in.readLine());                      15
      output = input.substring(start,start+len);                  16
      System.out.println("\nYour sub-string is \""+output+"\""); 17
  }                                                               18
}                                                                 19
```

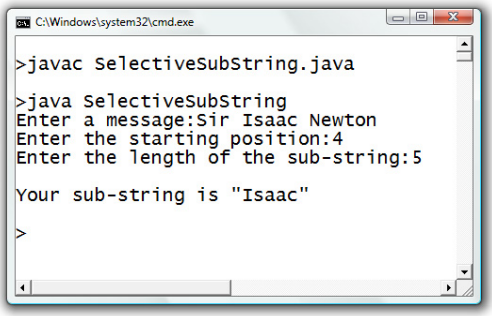


Figure 82: A program that reads two String inputs from the keyboard

The program asks the user to input a line of text as well as to pick a portion of the line by specifying the starting position together with the number of characters of that portion. The program uses *readLine()* to read in the input. On line 13 and line 15, the starting position and the length are converted to **`int`** values. The **`int`** values are then used with substring() correspondingly.

Example 36: Funny encoder

Let's look at the following Java program called FunnyEncoder.java. This program uses only what we have learnt so far. The program converts a 4-

digit string (E.g. 0345, 1829, etc.) into a specific code by mapping each digit to a specific funny pattern defined in the following table.

Digit	Pattern	Digit	Pattern
0	(^_^)	5	(^v^)
1	(-_-)	6	(^o^)
2	(>_<)	7	(^_____^)
3	(o_o)	8	(@_@)
4	(O_o)	9	(*__*)

Table 10: FunnyEncoder encoding scheme

For example, if the input digit string is 0123, the encoded string is **(-_-)(>_<)(o_o)(O_o)**. Here is the source code for the program and some example outputs. Make sure you go through the program and try to understand all of the statements.

```
import java.io.*;                                                    1
public class FunnyEncoder                                            2
{                                                                    3
    public static void main(String[] args) throws IOException       4
    {                                                                5
        int loc;                                                     6
        String input, output = "", s = "";                           7
        s += "(^_^)    ";                                            8
        s += "(-_-)    ";                                            9
        s += "(>_<)    ";                                            10
        s += "(o_o)    ";                                            11
        s += "(O_o)    ";                                            12
        s += "(^v^)    ";                                            13
        s += "(^o^)    ";                                            14
        s += "(^_____^)";                                            15
        s += "(@_@)    ";                                            16
        s += "( *__* ) ";                                            17
                                                                     18
        BufferedReader stdin = new BufferedReader(                   19
                   new InputStreamReader(System.in));                20
        System.out.print("Enter a 4-digit string:");                 21
        input = stdin.readLine();                                    22
        loc = 9*Integer.parseInt(input.substring(0,1));              23
        output += s.substring(loc,loc+9).trim();                     24
        loc = 9*Integer.parseInt(input.substring(1,2));              25
        output += s.substring(loc,loc+9).trim();                     26
        loc = 9*Integer.parseInt(input.substring(2,3));              27
        output += s.substring(loc,loc+9).trim();                     28
        loc = 9*Integer.parseInt(input.substring(3));                29
```

(continued on next page)

```
(continued from previous page)
        output += s.substring(loc,loc+9).trim();                    30
        System.out.println("Encoded String -> "+output);            31
    }                                                               32
}                                                                   33
```

Figure 83: The code listing of the FunnyEncoder program

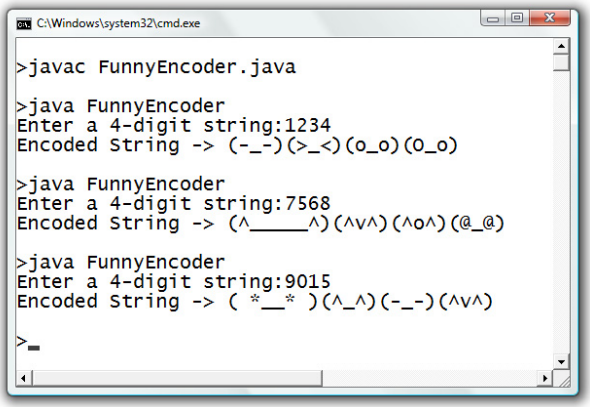


Figure 84: An output of the FunnyEncoder program

In this program, all patterns are concatenated into a long *String* object **s**, in the order from 0 to 9. Each pattern is padded with spaces so that every pattern spans 9 characters. The idea is to extract each digit from the input string and use the digit to locate the starting position of its corresponding pattern in **s** so that *substring()* can be used accordingly.

A *BufferedReader* object is used to read a line assumed to contain 4 digits. The starting position of the pattern associated with a digit i can be calculated based on the fact that the length of every pattern is fixed at 9 to be $9 \times i$. The position is stored in the variable **loc** and the encoded pattern of that digit can be found using **s.substring(loc,loc+9)**. The method *trim()* is used for getting rid of trailing whitespaces padded to each pattern.

Example 37: Vector decomposition

Now we wish to write a program that calculates the resulting force in the x and y directions, as illustrated in Figure 85, from the magnitude of the input force F (in Newton) and the angle between F and the x axis (in Degree).

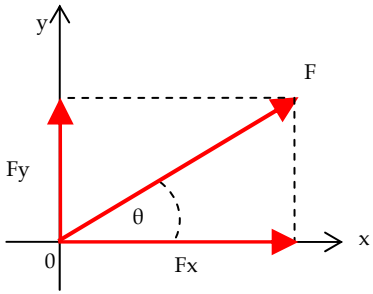


Figure 85: Decomposition of a vector F into Fx and Fy

Problem definition: The program needs to calculate the force in the x and y directions from the magnitude of the input force, F, and the angle, θ.

Analysis: There are two inputs, F and θ. Output, which are the force components in the two directions, are to be shown on screen.

Design:

- Prompt the user to input F, and store the input in **`f`**.
- Prompt the user to input θ, and store the input in **`theta`**.
- Convert θ, which is in degree, to radian by $\theta_{radian} = \theta_{degree} \times \frac{\pi}{180}$. Then, store the converted angle in **`thetaRad`**.
- Calculate the force component in the x direction from $F_x = F \cdot \cos(\theta_{radian})$. Then, store the result in **`fx`**.
- Calculate the force component in the y direction from $F_y = F \cdot \sin(\theta_{radian})$. Then, store the result in **`fy`**.
- Show the **`fx`** and **`fy`** on the screen.

Implementation:

```
import java.io.*;                                                    1
public class FindFComponents                                         2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
    double theta, f, thetaRad, fx, fy;                               6
    BufferedReader stdin =                                           7
      new BufferedReader(new InputStreamReader(System.in));          8
    // prompt for f                                                  9
    System.out.print("Enter the magnitude of F (Newton): ");       10
    f = Double.parseDouble(stdin.readLine());                       11
    // prompt for theta                                             12
    System.out.print("Enter the angle between ");                   13
    System.out.print("F and the x axis (Degree): ");                14
    theta = Double.parseDouble(stdin.readLine());                   15
    // convert degree to radian                                     16
    thetaRad = theta*Math.PI/180;                                   17
    // calculate fx and fy                                          18
    fx = f*Math.cos(thetaRad);                                      19
    fy = f*Math.sin(thetaRad);                                      20
    // show the results                                             21
    System.out.println("Fx = "+fx+" N");                            22
    System.out.println("Fy = "+fy+" N");                            23
  }                                                                 24
}                                                                   25
```

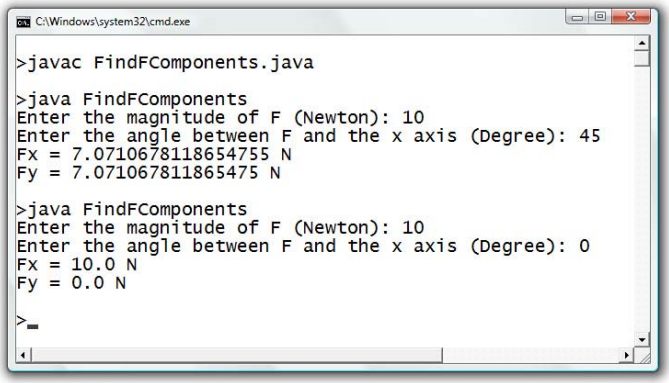


Figure 86: A program decomposing a force vector

This example shows a non-trivial program that utilizes the capability of reading inputs from the keyboard as well as how to perform Arithmetic

calculations covered in earlier chapters and how to handle conversion of *String* to **`double`** properly.

Reading Formatted Input using Scanner

Another alternative for reading input from keyboards is to use a class called *Scanner*, which is provided with Java in the *java.util package*. The class comes with methods that support reading *String* as well as input of primitive data types in formats (patterns) that can be defined.

Like *BufferedReader* as well as many other classes, in order to take the benefit from the static methods provided in the class, an object of the class is to be created first. Since *Scanner* is not in the default package, before we can use it in our program we have to import the class so that the compiler knows about it when the program gets compiled. The import statement is:

```
import java.util.Scanner;
```

To create an object of the class *Scanner* and have its input source bound with the keyboard, the following statement can be used.

```
Scanner sc = new Scanner(System.in);
```

This statement also creates a variable of the *Scanner* type and makes it refer to the new *Scanner* object whose source is bound to the standard input stream (I.e. the keyboard). Later on in the program, we can make use of the (non-static) methods provided by *Scanner* through this **`sc`** variable.

A *Scanner* object matches its *String* input obtained from its source with an input pattern that can be set. By default, the input pattern is a sequence of tokens separated by whitespaces, where each token can be thought of as a string of characters. The default pattern looks like the one in Figure 87.

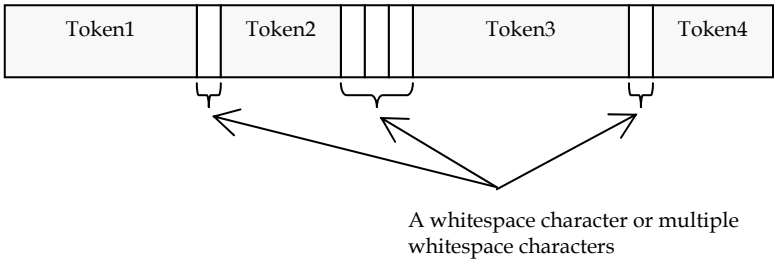


Figure 87: The default pattern used by a Scanner object

A *Scanner* object can read each token at a time and convert each token to its corresponding value in a specified primitive data type as well as *String* using various non-static method provided by the *Scanner* class.

To read a token from its input as a *String*, we can use the *next()* method, whose description is given below.

next()

When **`sc`** is a *Scanner* object, **`sc.next()`** considers the current input read by **`sc`**, finds the next token according to the defined pattern and parse it as a *String* object.

Note that when *next()* is called, it will cause the program to wait for the user's input if the *Scanner* object has not read any input before or when there are no more token to be parsed.

Example 38: Reading one String at a time using Scanner

The following Java program uses the *Scanner* class to read two tokens of input and use them as *String* objects.

```
import java.util.Scanner;                                          1
public class ScannerDemo1 {                                        2
    public static void main(String [] args){                      3
        Scanner sc = new Scanner(System.in);                       4
        System.out.print("Enter firstname and lastname:");        5
```

(continued on next page)

(continued from previous page)

```
        String fname = sc.next();                                    6
        String lname = sc.next();                                    7
        System.out.println("Firstname ="+fname);                      8
        System.out.println("Lastname  ="+lname);                      9
    }                                                                10
}                                                                    11
```

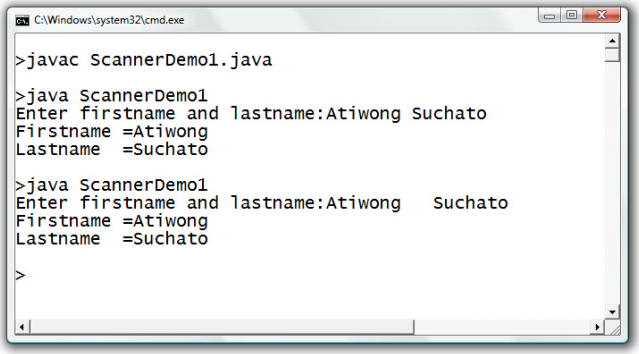


Figure 88: Using Scanner to read String objects

The above program declares and creates a *Scanner* object on line 4. Note that for the program to know about *Scanner*, we need the import statement as coded on the first line. The program prompts the user to input a first name and a last name via keyboard. The program assumes that the two pieces are separated using one or more whitespaces. When the *next()* method is invoked on line 6, the program waits for the user to type in a line of input. The method then tries to read the first token, all characters prior to the first occurrence of a whitespace, and assign it as a *String* object to **`fname`**. On line 7, *next()* is called again from **`sc`**. Here, the program looks at the line input previously received by **`sc`** and tries to parse another token and assign it as a *String* object to **`lname`**.

Figure 88 shows two executions of the program whose user inputs are different in number of whitespaces used to separate the two tokens. It works in both cases.

There are more methods than *next()* that let a *Scanner* object read a token from its input. These methods automatically convert tokens into different primitive data types.

Let's assume that sc is a Scanner object when considering the descriptions of the following methods.

nextBoolean()

`sc.nextBoolean()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `boolean` value.

nextByte()

`sc.nextByte()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `byte` value.

nextDouble()

`sc.nextDouble()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `double` value.

nextFloat()

`sc.nextFloat()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `float` value.

nextInt()

`sc.nextInt()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `int` value.

nextLong()

`sc.nextLong()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `long` value.

nextShort()

`sc.nextShort()` considers the current input read by `sc`, finds the next token according to the defined pattern and parse it as a primitive `short` value.

Example 39: Reading formatted data input

This example shows the use of *Scanner* to read in values of primitive data types as well as *String*. The program finds the average score from two data sets, each of which contains the name of a person and a numeric score.

```
import java.util.Scanner;                                          1
public class ScannerDemo2 {                                        2
    public static void main(String [] args){                      3
        String name1, name2;                                       4
        double score1, score2;                                     5
        Scanner sc = new Scanner(System.in);                       6
        System.out.println("--------------------------------");    7
        System.out.println("Enter each data set by specifying");   8
        System.out.println("a person name and his/her score  ");   9
        System.out.println("separated by spaces");                 10
        System.out.println("--------------------------------");    11
                                                                   12
        System.out.print("Data set 1:");                           13
        name1 = sc.next();                                         14
        score1 = sc.nextDouble();                                  15
                                                                   16
        System.out.print("Data set 2:");                           17
        name2 = sc.next();                                         18
        score2 = sc.nextDouble();                                  19
                                                                   20
        System.out.println("--------------------------------");    21
        System.out.print("The average score of "+name1+" and ");  22
        System.out.print(name2+" is "+(score1+score2)/2);          23
    }                                                              24
}                                                                  25
```

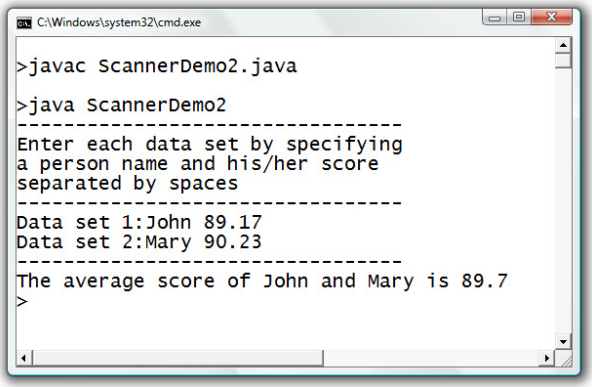


Figure 89: Using Scanner to read String objects and double values

The statements on line 4 to line 11 declare necessary variables, create a *Scanner* object, and show brief instructions to the user. The statements on line 13 to line 15 involve reading a *String* as well as a **`double`** value as two tokens read from **`sc`**. Here, the input must be in a format that starts with the name, follows with a single or multiple whitespaces, and then ends with a number. The statements on line 17 to line 19 perform the same thing with the second set of data. Once everything is read and assigned to the variables, the output can be processed accordingly.

More realistic examples will be shown later when we learn about decisions and iterations.

Exercise

1. Write valid Java statements that perform the following steps.
    a. Declare a variable for storing a *String*. Name it **`s1`**.
    b. Have **`s1`** refer to a new *String* object whose content is **`"Java"`**.
    c. Declare another variable named **`s2`** and have it refer to a new *String* object whose content is **`"Programming"`**.
    d. Print the concatenation of **`s1`** and **`s2`** on screen.

2. Explain in your own words the functionality of the two *dot* operators in the statement `System.out.print("I love eating!");`.

3. Given that *Calendar* is a valid class in Java and `c` is a variable referring to an object of the *Calendar* class, which of the following expression involve calling a method in the *Calendar* class. And, which ones simply access some data in the *Calendar* class. (Ignore their meanings for now.)

    a. Calendar.DECEMBER
    b. Calendar.getInstance()
    c. Calendar.getAvailableLocales()
    d. c.isTimeSet
    e. Calendar.MILLISECOND
    f. c.clear()
    g. c.get(1)

4. Write a Java program that calculates and shows the areas and circumferences of three circles, each of which has its radius of 3, 100, and 8.75 centimeters.

5. What is the output of the following code segment?

```
String s = "tachygraphometry";
System.out.println(s.charAt(1));
System.out.println(s.charAt(5));
System.out.println(s.charAt(12));
System.out.println(s.charAt(s.length()-1));
```

6. What is the output of the following code segment?

```
String s1 = "macaroni penguin";
String s2 = s1.substring(
    s1.indexOf(' ')+1,s1.length()).toUpperCase();
System.out.println(s1);
System.out.println(s2);
```

7. What is the output of the following code segment?

```
String s1 = "Houston";
String s2 = "Dallas".concat(s1);
s1 = s2.substring(2,4);
System.out.println(s1.length());
System.out.println(s2.length());
```

8. What is the output of the following code segment?

```
String s = "Jacobian";
System.out.println(s.indexOf('J'));
System.out.println(s.indexOf('c'));
System.out.println(s.indexOf('a'));
System.out.println(s.indexOf('j'));
System.out.println(s.indexOf('c'-1));
```

9. What is the output of the following code segment?

```
String s1 = "A";
String s2 = s1+1;
char c = 'A';
String s3 = c+1+"A";
System.out.println(s2.concat(s3).concat(s1));
```

10. What is the output of the following code segment?

```
String s = "1999";
System.out.println(String.valueOf(s));
System.out.println(String.valueOf(s)+1);
System.out.println(String.valueOf(s+1));
```

11. Explain why the *String* class is available to our program without the use of an *import* statement and why an *import* statement is required when we want to use the *BufferedReader* class in out program.

12. Write a Java program that prompts for and accepts a text message from the user via keyboard and prints it out on screen.

13. Write a Java program that prompts for two text messages from the user via keyboard, connect them together, and print the result on screen.

14. Write a Java program that prompts for and accepts a telephone number of the form xx-xxx-xxxx where each x is a digit (e.g. 02-123-9999), and prints it out in the following form: x-xxxx-xxxx (e.g. 0-2123-9999).

15. Write a Java program that prompts for and accepts an email address and prints the associated account's name and domain name in two separate lines. For example, me@somemail.com should be printed out as:

```
me
somemail.com
```

16. Write a Java program that prompts for and accepts two numbers, *a* and *b*, via keyboard, and prints out the results of the following numeric computation:

$$a+b,\ a\times b,\ \frac{a}{b},\ a^b,\ \text{and}\ \sqrt[b]{a}$$

Chapter 6: Decisions

Objectives

Students should

- Be able to use Java conditional constructs, including **`if`**, **`if-else`**, and **`switch`**, as well as the nested version of those constructs correctly.
- Be able to perform equality testing suitable with each primitive data type.
- Be able to design programs that require the inclusion of decisions.

Controlling the Flow of Your Program

Up to this point, you should be familiar with creating Java program that runs in straight lines, i.e. statements are executed in the order they are listed every time. Although this is enough for solving simple problems, it is not enough for general problem solving. Generally, we need to have control over which statements are executed and how often. In this chapter, we will look at Java's *conditional constructs* that control whether statements listed. Conditional constructs include **`if`**, **`if-else`**, and **`switch`**. In the next chapter, we will look at Java's *iterative constructs* that control how often statements are executed.

Before we look at conditional constructs, let's revisit FunnyEncoder.java presented in Example 36. The program converts a 4-digit string to its corresponding encoded string. The 4-digit string to be converted is supplied by the user of the program via keyboard. As long as the user enters any legal 4-digit string, the program works fine. However, the program is not fool-proof. What will happen if the length of the string entered is less than 4? Below are the results of inputting a 3-digit and 2-digit strings into the program.

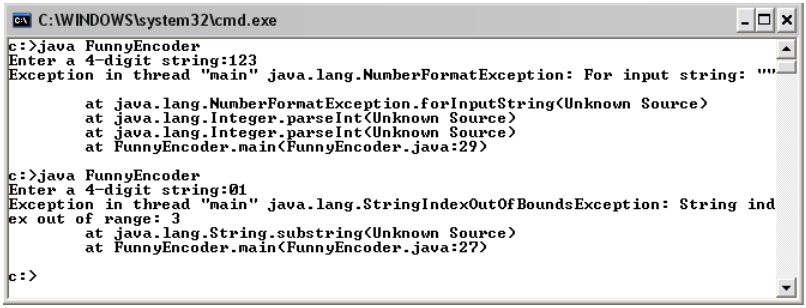


Figure 90: Run-time errors from the FunnyEncoder program without input validation

When the input is 123, which is a *String* of length 3, the program gives out an error because we try to find **`substring(3)`** on a *String* of length 3, in which case the index used for *substring()* is beyond the range of that *String*. The cause of the error, when the input is 01, is because of **`substring(2,3)`** which is due to the similar reason.

Naturally, we wish to have our program check for the length of the input digit string first, and then let the program process with the rest of the statements if the length equals 4. Otherwise, the program should prompt the user to input a new digit string with the valid length by executing a different set of statements. This is where conditional constructs comes into play.

We will visit FunnyEncoder.java again after we have learned about the syntax of Java's conditional constructs.

'If' Construct

When we want to write a program that involves the program flow shown below, i.e. **`Action`** is performed only when the **`Test Expression`** is **`true`**, we use an **`if`** statement.

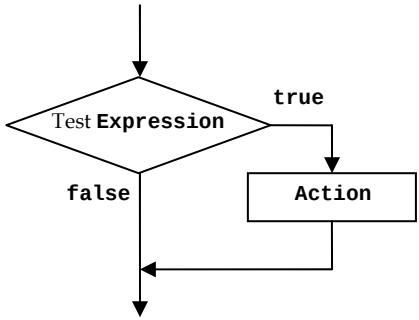


Figure 91: A flowchart associated with an if construct

A form of an **`if`** statement has the following syntax.

```
if (Test Expression) Action
```

`Test Expression` can be any expression that can be evaluated to a **`boolean`** value. **`Action`** can be a single Java statement or a series (or block) of statements enclosed in curly braces {}. Whether it is a single Java statement (E.g. **`j = k + 1;`**), or a block of statements (E.g. **`{j=k+1; k=0;}`**), it will be executed only when the **`boolean`** value of **`Test Expression`** is **`true`**. Otherwise, the program will skip over **`Action`** to the next statements, if there are any.

Consider the following examples.

Example 40: Implementing an absolute value finder

The following program shows how to use an **`if`** statement to check the sign of the input.

```
import java.io.*;                                                    1
public class ShowAbsolute                                            2
{                                                                    3
       public static void main(String[] args) throws IOException    4
       {                                                             5
           double d;                                                 6
```

(continued on next page)

(continued from previous page)

```
            BufferedReader stdin =                                  7
                new BufferedReader(new                              8
                    InputStreamReader(System.in));                  9
            System.out.print("Enter a number:");                     10
            d = Double.parseDouble(stdin.readLine());               11
                                                                    12
            if(d < 0) d = -d;                                       13
                                                                    14
            System.out.println("Its absolute value = "+d);          15
        }                                                           16
}                                                                   17
```

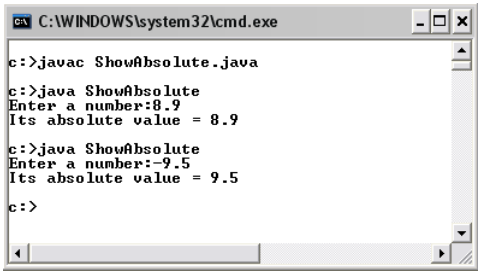


Figure 92: A program showing the absolute value of the input

This program gets a number from keyboard and shows the absolute value of that input. Line 13 uses an **`if`** statement to check whether **`d<0`**. If **`d<0`** is **`true`**, **`d = -d`** is executed. If **`d<0`** is false, **`d = -d`** is not executed and the program just continues on the next line. In this example, **`Action`** is a single Java statement **`d = -d;`** (which is terminated with a semicolon).

Example 41: Ad-hoc sorting of three inputs

Consider the following program where **`Action`** in the **`if`** statement is a block containing multiple statements.

```
import java.io.*;                                                   1
public class ShowInOrder                                            2
{                                                                   3
  public static void main(String[] args) throws IOException        4
  {                                                                 5
    double d1,d2,temp;                                              6
```

(continued on next page)

(continued from previous page)

```
    BufferedReader stdin =                                              7
      new BufferedReader(new InputStreamReader(System.in));            8
    System.out.print("Enter the 1 st. number:");                        9
    d1 = Double.parseDouble(stdin.readLine());                          10
    System.out.print("Enter the 2 nd. number:");                        11
    d2 = Double.parseDouble(stdin.readLine());                          12
                                                                        13
    if(d1 > d2){                                                        14
      temp = d1;                                                        15
      d1 = d2;                                                          16
      d2 = temp;                                                        17
    }                                                                   18
                                                                        19
    System.out.print("Showing the numbers ");                           20
    System.out.println("from small to large.");                        21
    System.out.println(d1+", "+d2);                                     22
  }                                                                     23
}                                                                       24
```

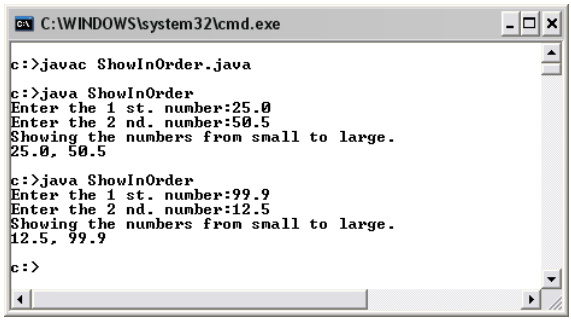


Figure 93: A program showing inputs from the smaller to the larger

This program takes two numbers from the users and showing them on screen in an ascending order. If you observe the code on line 22, you will see that no matter what numbers are input into **`d1`** and **`d2`**, we always print out **`d1`** and then **`d2`**. So, if **`d1`** is greater than **`d2`**, its value needs to be swapped. The swapping of the values are done on line 15, line 16, and line 17, all of these will be executed only if **`d1`** is greater than **`d2`**. That means if **`d1`** is originally smaller than **`d2`**, we do not have to do anything to the values prior to the printing out on line 22.

'If-else' Construct

Many times, when **`Test Expression`** is **`false`**, we do not want the program to just skip some portion of the code, but instead we want to execute a portion of the code that is different than when **`Test Expression`** is **`true`**. The mentioned situation can be depicted in the following picture. **`Action1`** will be done when Test Expression is **`true`**. **`Action2`** will be done when Test Expression is **`false`**.

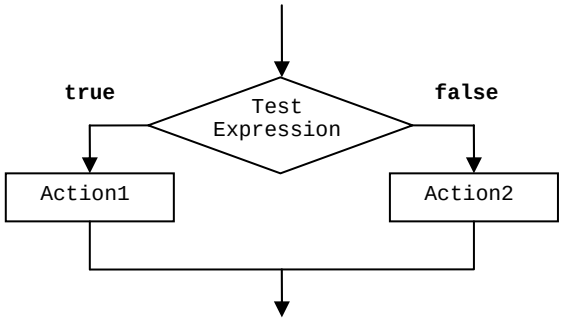


Figure 94: A flowchart associated with an if-else construct

The syntax of an **`if-else`** statement is of the following structure.

```
if(Test Expression) Action1 else Action2
```

Again, **`Test Expression`** can be any expression that can be evaluated to a **`boolean`** value. **`Action1`** and **`Action2`** can be single Java statements or series (or blocks) of statements enclosed in curly braces {}. Whether each of them is a single Java statement, or a block of statements, it will be executed based on the **`boolean`** value of **`Test Expression`**. After the **`if-else`** statement, the program will continue on to the next statements, if there are any.

Example 42: The bigger number

The following Java program prints the number that is bigger between two numbers input by the user.

```
import java.io.*;                                                    1
public class PrintBiggerNumber                                       2
{                                                                    3
  public static void main(String[] args) throws IOException          4
  {                                                                  5
      double d1,d2,bigger;                                           6
      BufferedReader stdin =                                         7
          new BufferedReader(new InputStreamReader(System.in));      8
      System.out.print("Enter the 1 st. number:");                   9
      d1 = Double.parseDouble(stdin.readLine());                     10
      System.out.print("Enter the 2 nd. number:");                   11
      d2 = Double.parseDouble(stdin.readLine());                     12
                                                                     13
      if(d1 < d2)                                                    14
          bigger = d2;                                               15
      else                                                           16
          bigger = d1;                                               17
                                                                     18
      System.out.println("The bigger number is "+bigger);            19
  }                                                                  20
}                                                                    21
```

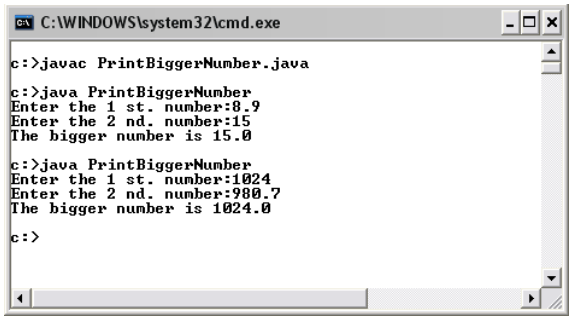


Figure 95: A program that use an if-else construct to check for the bigger input of the two inputs

d1 and **d2** are the two numbers input by the user. On line 14, the two numbers are compared. If **d1<d2** is **true**, line 15 is executed and consequently, the value in **d2** is stored in **bigger**. After that, the program proceeds on line 18 and so on. If **d1<d2** is **false**, line 17 is executed and consequently, the value in **d1** is stored in **bigger**. In a similar fashion, the program proceeds on line 18 and so on.

Example 43: Functions of x

The following program shows an example when **`Action1`** and **`Action2`** are blocks of statement. This program computes the value of *f(x)* and *g(x)* from these equations:

$$f(x) = \begin{cases} 2x; & 0 \le x \le 100 \\ 0; & \text{otherwise} \end{cases}$$

$$g(x) = \begin{cases} x^2; & 0 \le x \le 100 \\ 0; & \text{otherwise} \end{cases}$$

```
import java.io.*;                                                    1
public class FunctionsOfX                                            2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
    double x, f, g;                                                  6
    BufferedReader stdin =                                           7
        new BufferedReader(new InputStreamReader(System.in));        8
    System.out.print("Enter x:");                                    9
    x = Double.parseDouble(stdin.readLine());                        10
    if(x>=0 && x<=100){                                              11
        f = 2*x;                                                     12
        g = x*x;                                                     13
    }else{                                                           14
        f = 0;                                                       15
        g = 0;                                                       16
    }                                                                17
    System.out.println("f(x)="+f);                                   18
    System.out.println("g(x)="+g);                                   19
  }                                                                  20
}                                                                    21
```

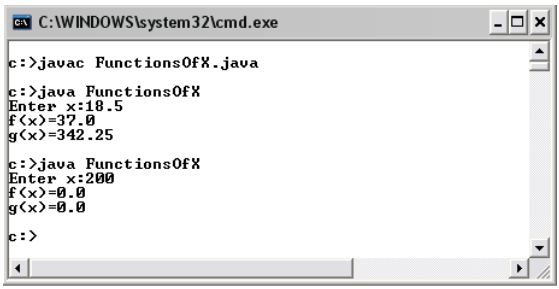


Figure 96: A program that use an if-else construct to choose which functions to be calculated

From line 11 to line 17, we can see that if `x>=0 && x<=100` is `true`, the block that contains `f = 2*x;` and `g = x*x;` is executed. Otherwise, the block that contains `f = 0;` and `g = 0;` is executed.

Nested If

Decision statements can be nested inside other decision statements. This should not come as a surprise to you since an `if` (and other decision constructs) statement is a valid java statement and any Java statements can be put inside an `if` statement (i.e. in the place of `Action` referred earlier). All you have to do is to think of an `if` statement in the same way as when you think of other Java statements.

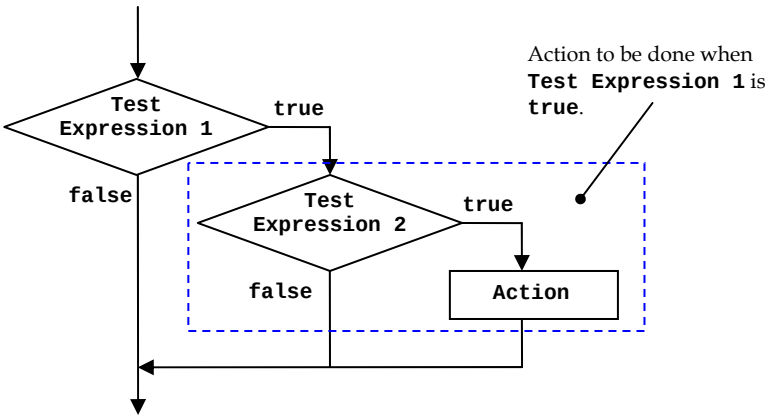


Figure 97: A flowchart associated with a nested if construct

For example, the above program flow can be written using the syntax shown in Figure 98.

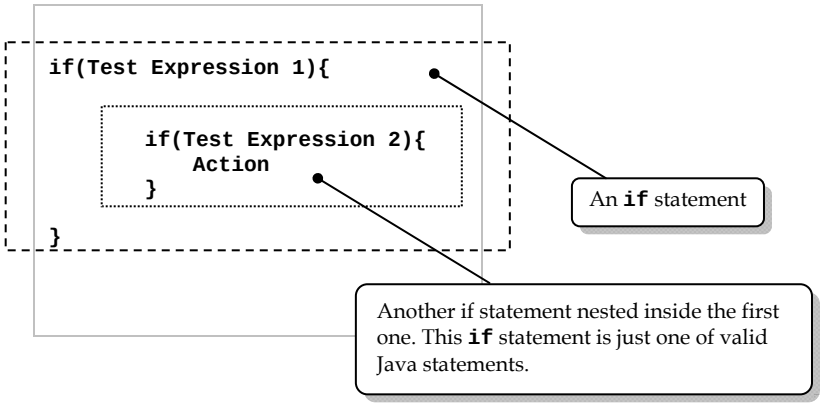


Figure 98: An example nested if statement

Let's consider the following example to see a little more complex nesting of **`if`** constructs as well as other additional statements.

Example 44: Nested conditions

Write Java statements corresponding to the flowchart shown in Figure 99. Assume that all variables and methods are valid.

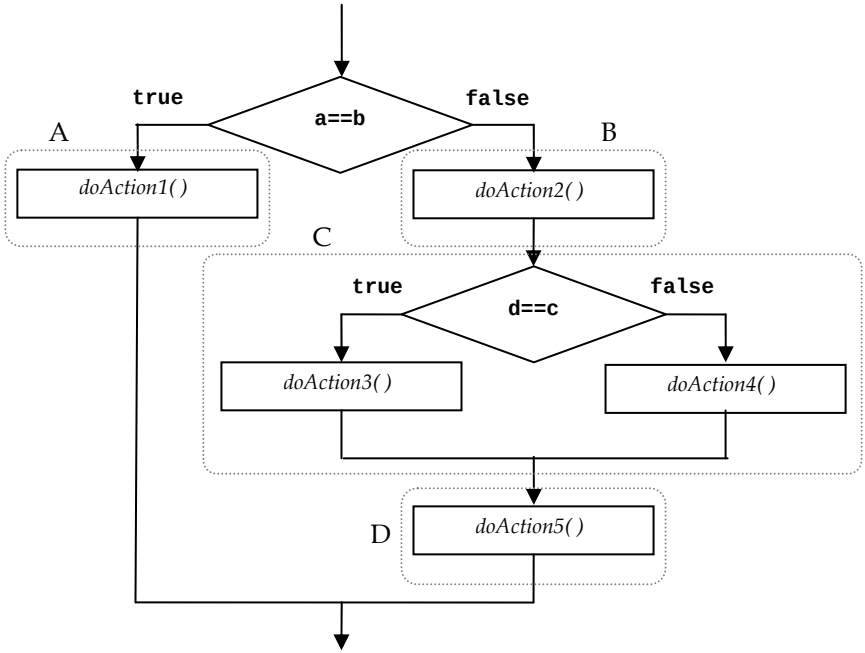


Figure 99: A flowchart representing a portion of a program

```
if(a==b){
    doAction1();                    A
}else{
    doAction2();                    B
    if(d==c){                       C
        doAction3();
    }else{
        doAction4();
    }
    doAction5();                    D
}
```

Figure 100: A flowchart representing a portion of a program

The code associated with the above program flow could take the form shown in the following figure. Note that the letters A to D labeled to the dotted shapes are used to associate parts of the flowchart with their associated portions of the code.

'If-else-if' Construct

When the action to be done in an **`else`** case consists of only a nested **`if`** statement, curly braces can be omitted, just like the case where the action consists of only one of any Java statements.

For example, the following statement

```
if(Test Expression 1){
    Action1
}else{
    if(Test Expression 2){
            Action2
    }else{
            if(Test Expression 3){
                    Action3
            }
    }
}
```

can also be written as,

```
if(Test Expression 1)
    Action1
else if(Test Expression 2)
    Action2
else if(Test Expression 3)
    Action3
```

The construct used just above is sometimes referred to as an **`if-else-if`** construct. However, readers should think about it as just a way to write some special cases of ordinary **`if-else`** statements.

Example 45: If neither one is bigger, they are equal

Recalling the example of PrintBigNumber.java presented earlier in this chapter, the program in the example prints on screen the bigger number of the two numbers input by the user. Now we will modify the program using a nested **if** statement so that the program handles the case where the two numbers are equal better.

```
import java.io.*;                                                    1
public class PrintBiggerNumber2                                     2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
    double d1,d2;                                                    6
    BufferedReader stdin =                                           7
        new BufferedReader(new InputStreamReader(System.in));        8
    System.out.print("Enter the 1 st. number:");                    9
    d1 = Double.parseDouble(stdin.readLine());                       10
    System.out.print("Enter the 2 nd. number:");                    11
    d2 = Double.parseDouble(stdin.readLine());                       12
                                                                     13
    if(d1 < d2)                                                      14
        System.out.println("The bigger number is "+d2);              15
    else if(d1 > d2)                                                 16
        System.out.println("The bigger number is "+d1);              17
    else                                                             18
        System.out.println("The two numbers are equal.");            19
                                                                     20
  }                                                                  21
}                                                                    22
```

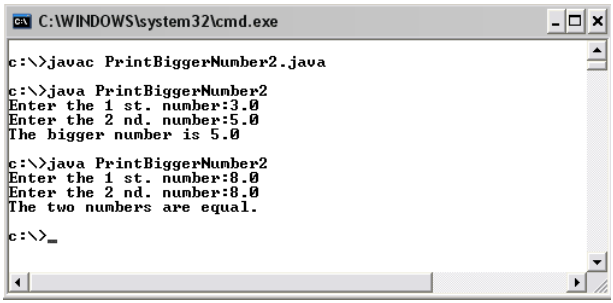


Figure 101: A program that prints out the comparison between two inputs using an if-else-if construct

Example 46: Funny encoder revisited

Now, we revisit FunnyEncoder.java again. We left it that we would like to check whether the user input a digit string of length four or not.

```
import java.io.*;                                                    1
public class FunnyEncoder2                                           2
{                                                                    3
        public static void main(String[] args) throws IOException   4
        {                                                            5
            int loc;                                                 6
            String input, output = "", s = "";                       7
            s += "(^_^)    ";                                        8
            s += "(-_-)    ";                                        9
            s += "(>_<)    ";                                        10
            s += "(o_o)    ";                                        11
            s += "(O_o)    ";                                        12
            s += "(^v^)    ";                                        13
            s += "(^o^)    ";                                        14
            s += "(^_____^)";                                        15
            s += "(@_@)    ";                                        16
            s += "( *__* ) ";                                        17
                                                                     18
            BufferedReader stdin =                                   19
                new BufferedReader(                                  20
                    new InputStreamReader(System.in));               21
                                                                     22
            System.out.print("Enter a 4-digit string:");             23
            input = stdin.readLine();                                24
                                                                     25
            //Validate input string                                  26
            int len = input.length();                                27
            if(len != 4){                                            28
                System.out.println("Input must be of length 4!");    29
            }else{                                                   30
                loc = 9*Integer.parseInt(input.substring(0,1));      31
                output += s.substring(loc,loc+9).trim();             32
                loc = 9*Integer.parseInt(input.substring(1,2));      33
                output += s.substring(loc,loc+9).trim();             34
                loc = 9*Integer.parseInt(input.substring(2,3));      35
                output += s.substring(loc,loc+9).trim();             36
                loc = 9*Integer.parseInt(input.substring(3));        37
                output += s.substring(loc,loc+9).trim();             38
                System.out.println("Encoded String -> "+output);     39
            }                                                        40
        }                                                            41
}                                                                    42
```

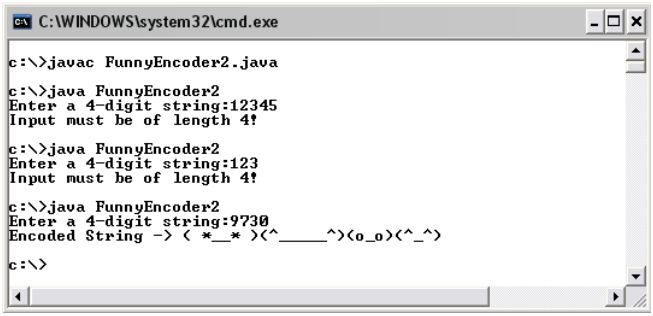


Figure 102: FunnyEncoder with input length checking

Here, we add an **`if`** statement on line 28 in order to check the validity of the input in terms of its length. The **`if`** statement decides whether to proceed with converting each digit in the input into the given patterns or not.

Furthermore, we should also check whether each character in the input *String* is one of the ten digits or not. It is intentionally left as an exercise for curious readers.

Use Braces to Avoid Coding Confusions

It is strongly recommended to always use curly braces even the action to be done in each case consists of only one statement in order to avoid confusions. Consider the following code segment.

```
if(p)
    System.out.println("A");
    if(q)
        System.out.println("B");
else{
    System.out.println("C");
}
```

Let's say the question is what the program will print out when **`p`** is **`false`** and **`q`** is **`true`**. The answer is that "**`B`**" will be printed out. If your answer is

different, you should pay attention to which **if** statement each action belongs to. Do not let the indentation fool you. An equivalent code segment can be written using proper curly braces as:

```
if(p){
    System.out.println("A");
}
if(q){
    System.out.println("B");
}else{
    System.out.println("C");
}
```

This is easier to read and less likely to generate confusions.

The ? : Operator

An **if** statement can sometimes be replaced by the ? : operator, which has the following form.

```
Test Expression ? Expression1 : Expression2
```

A code segment of the above form is an expression. As a reminder, an expression must be able to be evaluated to a value. The value of this expression depends on the value of **Test Expression**. If **Test Expression** is **true**, the value of the whole expression is equivalent to the value of **Expression1**. If **Test Expression** is **false**, the value of the whole expression is equivalent to the value of **Expression2**.

For example, the statement:

```
int bigger = (intA > intB) ? intA : intB;
```

is equivalent to:

```
int bigger;
if(intA > intB){
    bigger = intA;
else{
    bigger = intB;
}
```

Example 47: The absolute value (again! but with short-handed expression)

The following program works similarly to the ShowAbsolute program in Example 40. However, the ? : operator is used in the place of the **if** construct.

```
import java.io.*;                                                    1
public class ShowAbsoluteShort {                                     2
    public static void main(String [] args) throws IOException{     3
        BufferedReader in =                                          4
                new BufferedReader(                                  5
                    new InputStreamReader(System.in)                 6
                );                                                   7
        System.out.print("Enter a number:");                        8
        double d = Double.parseDouble(in.readLine());                9
        System.out.print("Its absolute value = "+(d<0?-d:d));        10
    }                                                                11
}                                                                    12
```

Figure 103: A program finding the absolute value of the input in which ?: is used instead of the full if construct

Reader should pay attention to the expression **d<0?-d:d** on line 10. The value of this expression is **-d** if **d<0** is true, otherwise it is just **d**.

Equality Testing for Values of Primitive Data Types

When comparing two values to test whether they are equal, we need to take their data types into account. Using variables with the == operator compares the values stored in those variables. Using == to compare values of primitive data types usually works fine, except for floating points. Consider the following example.

Example 48: Equality testing

The EqualityTesting program listed below shows examples of expressions comparing values of variables of primitive data types. The equality operator == is used for the comparison.

```
public class EqualityTesting                                          1
{                                                                     2
       public static void main(String[] args)                        3
       {                                                              4
           //comparing int                                           5
           int a, b;                                                  6
           a = 1;                                                     7
           b = 1;                                                     8
           System.out.print("a  and b  are “);                       9
           System.out.println((a==b)?"equal.":"not equal.");          10
           //comparing char                                          11
           char c1, c2;                                               12
           c1 = 'z';                                                  13
           c2 = 'z';                                                  14
           System.out.print("c1 and c2 are ");                       15
           System.out.println((c1==c2)?"equal.":"not equal.");        16
           //comparing double                                        17
           double d1, d2;                                             18
           d1 = 1.44;                                                 19
           d2 = 1.44;                                                 20
           System.out.print("d1 and d2 are ");                      21
           System.out.println((d1==d2)?"equal.":"not equal.");        22
           //comparing double                                        23
           double d3, d4;                                             24
           d3 = 0.9;                                                  25
           d4 = 0.3+0.3+0.3;                                          26
           System.out.print("d3 and d4 are ");                      27
           System.out.println((d3==d4)?"equal.":"not equal.");        28
           System.out.println("d3="+d3);                              29
           System.out.println("d4="+d4);                              30
       }                                                              31
}                                                                     32
```

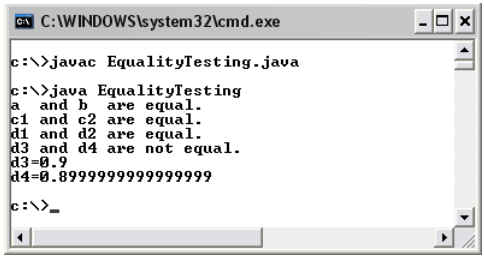


Figure 104: Demonstration of using == to compare values of primitive data types

We have found that **a** and **b** are equal, **c1** and **c2** are equal, as well as **d1** and **d2** are equal as expected. However, **d3** and **d4** are not while they should. The reason lies in the limitation in representing floating points with binary representation as discussed in Chapter 4. Thus, we have to take this into account when comparing floating point values.

Safe Ways to Compare Floating Point Values

To compare whether two floating point values are equal, we instead test whether the difference between them are small enough. Thus, two values, x and y, are said to be equal as long as $|x - y| < \varepsilon$, where ε is a small tolerance value. Consider the following example.

Example 49: Floating-point value comparison

The following program shows a better way to compare two floating point values than using the equality operator.

```
public class EqualityTesting2                                          1
{                                                                      2
       public static void main(String[] args)                         3
       {                                                               4
           double d3, d4;                                              5
           final double MAX_DIFF = 1e-10;                              6
           d3 = 0.9;                                                   7
```

(continued on next page)

(continued from previous page)

```
            d4 = 0.3+0.3+0.3;                                          8
            System.out.println("d3="+d3);                              9
            System.out.println("d4="+d4);                              10
            boolean isEqual =                                          11
                (Math.abs(d3-d4)<MAX_DIFF)? true : false;              12
            System.out.print("d3 and d4 are ");                       13
            System.out.println(isEqual?"equal.":"not equal.");         14
        }                                                              15
}                                                                      16
```

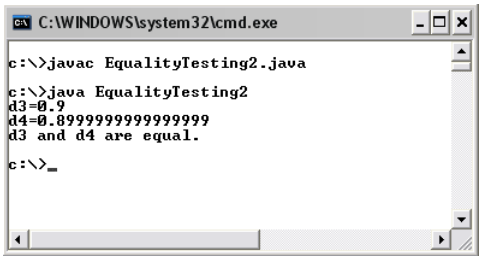


Figure 105: Demonstration of using == to compare values of primitive data types

In this example, **`d3`** and **`d4`** are considered equal if the difference between them is less than 10^{-10}.

From Example 49, whether the way we compare the values is considered working depends on the context of the program. In the example, the maximum allowed difference is at 10^{-10} which seems to be okay when comparing values in the proximity of 0.9. However, let's think about the situation in which we would like to compare values of much smaller magnitudes. Consider the answer to this question: "What is the value of $(8\times10^{-11})/4$?" We can simply tell that the answer is 2×10^{-11}. Then, if a program calculates $(8\times10^{-11})/4$ as 1×10^{-11}, will you consider it as a correct answer? The answer should be "No." Now, let's consider if we use the similar logic as presented in Example 49 (which is "**`d3`** and **`d4`** are considered equal if the difference between them is less than 10^{-10}") in comparing 2×10^{-11} and 1×10^{-11}. Such a comparison will results in that both values are considered equal since the difference between them is less

than 10^{-10}. This comparison does not deliver the desired result. To avoid such a problem, ones need to be careful about the threshold difference used in the program.

A neat trick normally used to avoid finding the threshold difference that works well with values of all magnitudes is to consider the amount of the difference relative to the size of the compared values. That means, if the compared values are large, the allowed difference should be large and if they are small, the allowed difference should also be small. For example, one could use the following test to compare whether *a* and *b* are considered equal.

$$\frac{|a-b|}{\max(|a|,|b|)} \times 100 < \varepsilon$$

The expression $\max(|a|,|b|)$ represents the bigger value between the absolute value of *a* and the absolute value of *b*. This test suggests that if the absolute difference between the values *a* and *b* is smaller than ε % of $\max(|a|,|b|)$, both values will be considered equal. Otherwise, they are not.

Equality Testing for Non-Primitive Data Types

Recall that what is stored in a variable of any non-primitive data types, or class, is a reference to an object of that class. Therefore, when the relational equality (==) is used to compare two variables of non-primitive data types, it compares whether the references are identical. In other words, the relational equality will yield **`true`** if both variables contain exactly the same object, not just different objects that might have identical properties, or, more precisely, attributes. Equality testing for objects of non-primitive data types with identical attributes can be performed by using the *equal()* method.

The expression **`a.equals(b)`** where **`a`** is a variable referring to an object and **`b`** is a variable referring to another object is evaluated as **`true`** if both objects have identical properties without having to be the same object. Otherwise, it is evaluated to **`false`**. The case when two *String* objects with

their contents containing the same character sequences is an example case of when two objects have identical properties.

Note that the default implementation of *equals()* is *reflexive* meaning that **`a.equals(b)`** should be evaluated to the same value as **`b.equals(a)`**.

String Equality Testing

Just like other non-primitive data types, if a relational equality is used to compare two variables containing *String* objects, it will return **`true`** if and only if the two variables contain references to the same *String* object. Consider the following code segment.

```
String s1 = "Espresso";
String s2 = "press";
System.out.println(s1.substring(2,7)==s2);
```

The output shown will be **`false`** even though the content of **`s1.substring(2,7)`** and **`s2`** are both "press". The reason is simply because both *String* objects are different objects.

Still, in Java, there are cases where it seems like two variables refer to different *String* objects but, underlyingly, they in fact refer to the same object. To see this, let's consider the following code segment.

```
String s1 = "Viva Java";
String s2 = "Viva Java";
System.out.println(s1==s2);
```

The output of the code segment is **`true`**. With just the explanation described above, it might be surprising since it seems like s1 and s2 are assigned with two different *String* objects. This can be explained based on the implementation of *String* in Java, in which *String* objects are *immutable* objects. However, such a concept like immutable objects will not be covered in this book and it should not be of concern at all for beginners wishing to learn how to write Java programs.

Therefore, what you as a reader of this book should know is a suitable way to compare *String* objects in Java without having to worry about

such unnecessary issues and that suitable way of comparing *String* object is to use the *equals()* method.

Both the **boolean** variables **p** and **q** in the following code segment are **true**.

```
String s1 = "Viva Java";
String s2 = "Viva Java";
boolean p = s1.equals(s2);
boolean q = s2.equals(s1);
```

Another method that involves *String* comparison is called *compareTo()*.

compareTo()

Given that **s1** and **s2** are String objects. The expression **s1.compareTo(s2)** returns:

- **0** if **s1** and **s2** have the same character sequence.
- **s1.charAt(k)-s2.charAt(k)** if there is a smallest position **k**, at which they differ.
- **s1.length()-s2.length()** if there is no position **k** at which they differ.

Example 50: Comparing Strings with compareTo()

Observe the output of the following program.

```
public class StringCompareDemo1                                          1
{                                                                        2
  public static void main(String[] args)                                3
  {                                                                      4
      System.out.println("Wonderland".compareTo("Wonderland"));          5
      System.out.println("Wonderful".compareTo("Wonderboy"));            6
      System.out.println("Wonder".compareTo("Wonderboy"));               7
      System.out.println("Wonderful".compareTo("Wonderland"));           8
  }                                                                      9
}                                                                        10
```

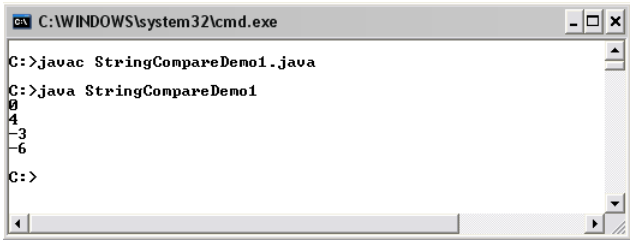


Figure 106: Demonstration showing compareTo() in action

The result from line 5 is **0** since both strings contain the same text. The result from line 6 is 4 since the characters at the smallest position that both strings differ are '**f**' and '**b**'. Thus, the result equals '**f**'-'**b**'. The result from line 7 is -3 since there is no position that the two strings differ. Thus, the result is the difference in their length. The result from line 8 is -6 due to the same reason since -6 is '**f**'-'**l**'.

'Switch' Constructs

It is common that decisions on which code segments will be executed are determined on the value of a variable. Java provides a conditional construct that selects which code segment to be executed from a number of them based on an integer value, E.g. a value of type **int**, **char**, etc. This construct is the **switch** construct, which has the following form.

```
switch(SwitchExpression){
    case CaseExp1 :
            Action1
            break;
    case CaseExp2 :
            Action2
            break;
            :
            :
            :
    case CaseExpN :
            ActionN
            break;
    default :
            DefaultAction
}
```

When a **switch** statement is executed, **SwitchExpression** is evaluated. Then, the program flow jumps to the case whose associated expression **CaseExp** equals to the value of **SwitchExpression**. if the value of **SwitchExpression** does not match with any **CaseExp**'s, the program flow jumps to **default**. After that, the code below that point will be executed in a regular fashion until a **break** command or the end of the **switch** block is reached. When a **break** is reached, the program flow jumps to the closing braces of that **switch** statement immediately. The program flow corresponds to such a **switch** statement can be illustrated in Figure 107.

One thing to keep in mind is that if there are no **break** statements, the program will not jump right to the end of the **switch** statement. It will just execute the statements along its way from top to bottom until it goes out of the **switch** statement naturally.

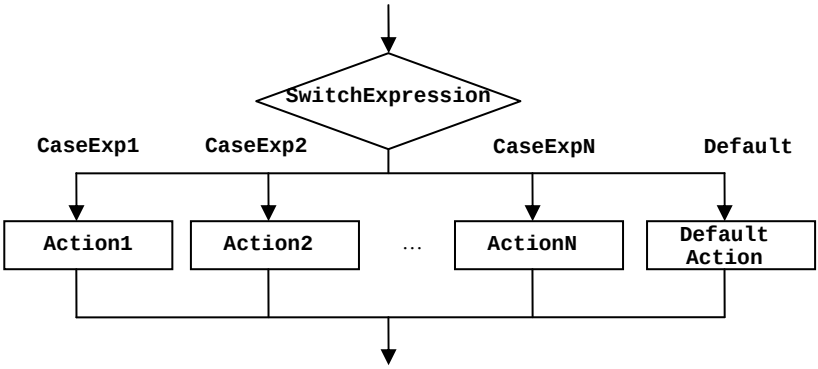


Figure 107: A flowchart representing a switch construct

Example 51: Printing Stars

Let's look at the following code.

```
import java.io.*;                                                    1
public class SwitchDemo                                              2
{                                                                    3
       public static void main(String[] args) throws IOException    4
       {                                                             5
           int n;                                                    6
```

(continued on next page)

(continued from previous page)

```
            String stringToPrint;                                   7
            BufferedReader stdin =                                  8
                new BufferedReader(                                 9
                    new InputStreamReader(System.in));              10
            System.out.print("Enter an integer from 1 to 4:");      11
            n = Integer.parseInt(stdin.readLine());                 12
            switch(n){                                              13
                case 1:                                             14
                    stringToPrint = "*";                            15
                    break;                                          16
                case 2:                                             17
                    stringToPrint = "* *";                          18
                    break;                                          19
                case 3:                                             20
                    stringToPrint = "* * *";                        21
                    break;                                          22
                case 4:                                             23
                    stringToPrint = "* * * *";                      24
                    break;                                          25
                default:                                            26
                    stringToPrint = "Integer out of range.";        27
            }                                                       28
            System.out.println(stringToPrint);                      29
        }                                                           30
}                                                                   31
```

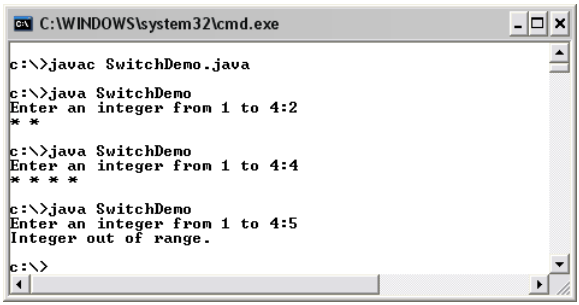


Figure 108: A program printing * whose number is determined using a switch construct based on the keyboard input

This program receives an integer from the user and stores that integer in **n**, on line 11. The **switch** statement checks the value of **n**. Suppose n equals 2, the program jumps to line 17 and then continues the execution of line 18. The program reaches a **break** statement on line 19 which makes the program jumps out of the **switch** statement onto line 28.

Now consider the following program, which is rather similar to the previous one. However, you should notice that the **`break`** statements are intentionally omitted from the **`switch`** construct.

```
import java.io.*;                                                    1
public class SwitchDemo2                                            2
{                                                                    3
       public static void main(String[] args) throws IOException    4
       {                                                             5
           int n;                                                    6
           BufferedReader stdin =                                    7
               new BufferedReader(                                   8
                   new InputStreamReader(System.in));                9
           System.out.print("Enter an integer from 1 to 4:");       10
           n = Integer.parseInt(stdin.readLine());                  11
           switch(n){                                               12
               case 1:                                              13
                 System.out.println("*");                           14
               case 2:                                              15
                 System.out.println("* *");                         16
               case 3:                                              17
                 System.out.println("* * *");                       18
               case 4:                                              19
                 System.out.println("* * * *");                     20
               default:                                             21
                 System.out.println("Integer out of range.");       22
           }                                                        23
       }                                                            24
}                                                                   25
```

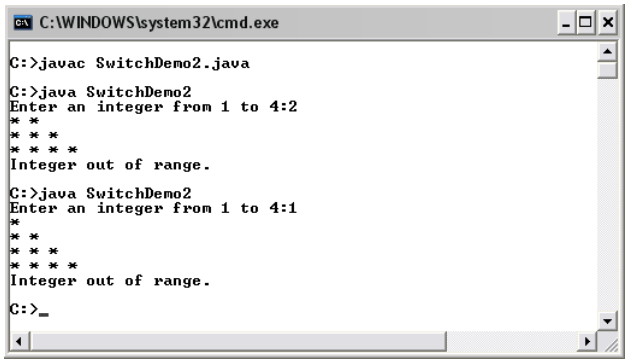


Figure 109: Demonstration of a program that the break statements are intentionally omitted from the switch construct

Observe the screenshot shown in Figure 109, we can see that once the program jumps to the case corresponding to the input, it continues executing statements that are followed regardless of the occurrences of the **case** keyword. When not using the **break** statement in some cases of the **switch** construct, make sure that it is really what you want your program to do.

Common Instructions for Multiple Cases

If we would like more than one case to execute the same set of statements, we can put the cases together. For example, we can write:

```
switch(n){
    case 1:
            ActionA
            break;
    case 2: case 3:
            ActionB
            break;
    case 4:
            ActionC
            break;
    case 5: case 6:
            ActionD
            break;
    default:
            ActionE
}
```

to have **ActionA** performed when **n** is 1, **ActionB** performed when **n** is 2 or 3, **ActionC** performed when **n** is 4, **ActionD** performed when **n** is 5 or 6, and **ActionE** performed when **n** is a number other than those that we have mentioned.

Apart from data types for integers, a value of **char** type can also be used as the condition in the switch statement. For example, if the variable **grade** is of type **char**, we can write:

```
switch(grade){
    case 'A': case 'a': case 'B': case 'b':
            Action1
            break;
```

```
        case 'C': case 'c': case 'D': case 'd':
                Action2
                break;
        case 'F': case 'f':
                Action3
                break;
        default:
                Action4
    }
```

to have **`Action1`** performed when **`grade`** is 'A', 'a', 'B', or 'b', **`Action2`** performed when **`grade`** is 'C', 'c', 'D', or 'd', **`Action3`** performed when **`grade`** is 'F', or 'f'. Otherwise, **`Action4`** is performed.

Example 52: Number base converter

A student is studying number systems and needs to convert integers found in real life to its base 2, base 8, and base 16 representations. Write a Java program that could be useful to him in this case.

Problem defining: Write a Java program that converts an input integer to either its binary (base 2), octal (base 8), or hexadecimal (base 16) representation.

Analysis: The input should be entered from the keyboard as decimal integers only. The user can make a choice whether to convert the input into which base system. The program should prompt the user about the choices he can make. The choice is indentified by typing in the letter 'B' for binary, 'O' for octal, and 'H' for hexadecimal. Cases of the letter will be ignored. "Invalid choice" will be shown on screen if the choice is not valid in a way and the program shall terminate normally. The result will be shown on screen.

Design: A *BufferedReader* object will be used to read in a line of input each time the program prompts the instruction to the user. We will assume that the user is co-operative and always makes a valid integer input. The program will perform just a simple input validity checking for the user's choice of the destination base system. When the choice is chosen, the length of the input has to be one character. We will also use the **`default`** case of a **`switch`** statement to handle a choice that is other than 'B', 'b', 'O', 'o', 'H', and 'h'. The **`switch`** statement will handle these valid choices

accordingly. Three static methods provided by the *Integer* class including *Integer.toBinaryString(), Integer.toOctalString(),* and *Integer.toHexString()* will be used to create *String* objects whose contents are the representation in binary, octal, and hexadecimal systems of the **`int`** value supplied to the methods respectively.

For the switch construct, the cases for 'B' and 'b' are listed together since they bear the same meaning as we decide to ignore cases of the letter. Same is applied to the cases for 'O' and 'o' as well as the cases for 'H' and 'h'.

Implementaion: The source code of the program can be observed in Figure 110.

```
import java.io.*;
public class BaseConverter
{
       public static void main(String[] args) throws IOException
       {
           String base, output;
           int input;
           BufferedReader stdin =
               new BufferedReader(
                 new InputStreamReader(System.in));
           System.out.print("Enter an integer in base 10:");
           input = Integer.parseInt(stdin.readLine());
           System.out.println("Convert "+input+" to?");
           System.out.println("----------------------");
           System.out.println("B or b for binary");
           System.out.println("O or o for octal");
           System.out.println("H or h for hexadecimal");
           System.out.println("----------------------");
           System.out.print(":");
           base = stdin.readLine();
           if(base.length()!=1){
               System.out.println("Invalid choice.");
           }else{
               switch(base.charAt(0)){
                 case 'B': case 'b':
                   output = Integer.toBinaryString(input);
                   break;
                 case 'O': case 'o':
                   output = Integer.toOctalString(input);
                   break;
```

(continued on next page)

(continued from previous page)

```
                case 'H': case 'h':
                  output = Integer.toHexString(input);
                  break;
                default:
                  output = "Invalid choice.";
              }
              System.out.println(output);
          }
      }
}
```

Figure 110: Demonstration of a program that the break statements are intentionally omitted from the switch construct

Testing: Some screenshots of the output of the program is shown in Figure 111.

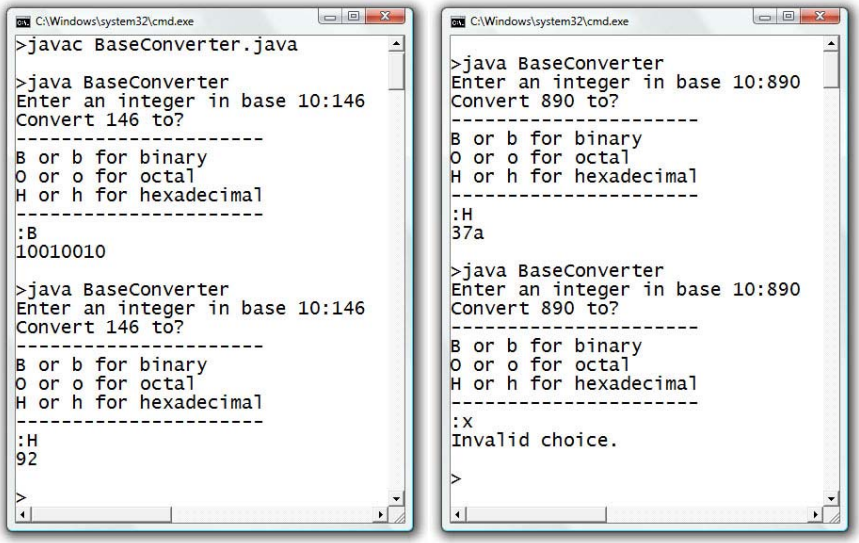


Figure 111: A program converting an integer in the decimal system to binary, octal, or hexadecimal system

Exercise

1. Assume that x and y are valid **int** variables. Consider the following code segment:

```
if(x!=y){
    System.out.println("1");
}
if(x>y){
    System.out.println("2");
}
if(x%y == 0){
    System.out.println("3");
}
```

   What is the output if:

   i. x = 2, y = 6
   ii. x = 1, y = 1
   iii. x = 9, y = 4
   iv. x = 10, y = 5

2. Assume that p and q are valid **boolean** variables. Consider the following code segment:

```
if(p && q){
    q = false;
}else{
    if(!q)
        System.out.println(p);
    if(p == q)
        System.out.println(p||q);
}
System.out.println(q);
```

   What is the output if:

   i. p = true, q = true
   ii. p = true, q = false
   iii. p = false, q = true
   iv. p = false, q = false

3. Assume that x, y and z are valid **int** variables. Consider the following code segment (note the poor indentations):

```
if(x>y||z>y)
    System.out.println("1");
else
    System.out.println("2");
if(Math.abs(x-y)>=z)
if(x>y)
    System.out.println("3");
else
    System.out.println("4");
else
    System.out.println("5");
```

What is the output if:

i. x = 1, y = 1, z = 1
ii. x = 2, y = 1, z = 0
iii. x = 3, y = 5, z = 4

4. Write a code segment that prints the value of an **int** variable k unless the value is less than 6.

5. Write a code segment that sets integer **d** to 1 if the integer **a** is less than or equal to 5 while integer **b** is not bigger than the difference between **a** and another integer **c**. Otherwise, set **d** to 0 if **c** is 0.

6. When **a** and **b** are two double variables, consider the following code segment.

```
double tol = 1e-25;
double x = (a*b)/(b-a);
double y = Math.sqrt(a/b);
boolean p = Math.abs(a-b)/Math.max(a,b)>tol;
boolean q = (a>b)||(b>x);
if(p||q == y>x){
    System.out.println("BLUE");
}else{
    System.out.println("RED");
}
```

Give the ranges of values for **a** and **b** that cause the code segment to display both "BLUE" and "RED". If no ranges can be found, explain why?

7. Convert the following code into a *switch* statement, when **k** contains an **int** value.

```
String cmd;
if(k==1){
    cmd = "Edit";
}else if(k==2){
    cmd = "Add";
}else if(k==3){
    cmd = "Quit";
}else{
    cmd = "Invalid";
}
```

8. Write a statement that set a *String* **s** to "Odd" if an integer k is an odd number and set **s** to "Even" if **k** is even. Using:
   i. an if-else statement.
   ii. a switch statement.

9. Convert the following code into a *switch* statement, when **k** contains an **int** value.

```
int p;
if(k==1||k==3){
    p = 1;
}else if(k==2||k==4){
    p = 2;
}else if(k==5){
    p = 3;
}else{
    p = 4;
}
```

10. What are the outputs of the following code segment when k = 0, 1, 2, 3, 4.

```
switch (k){
    case 1:
        System.out.println("A");
    case 2: case 3:
        System.out.println("B");
        break;
    case 4:
        System.out.println("C");
    default:
        System.out.println("D");
}
```

11. What is the output of the following code segment?

```
int a = 1, b = 2;
System.out.println(a>b?a:b);
```

12. What are the **boolean** values of **p** and **q** that make the output of the following code segment **true**?

```
System.out.println(p!=q&&!p?p:q);
```

13. Re-write the following code segment using ?: operators. Give that **c** contain a **char** value.

```
boolean p;
if(c=='a'){
    p = true;
}else{
    p = false;
}
```

14. Re-write the code in the previous problem without using any conditional constructs.

15. Re-write the following code segment using ?: operators. Give that **n** contain an **int** value.

```
if(n==0){
    System.out.println("Zero");
}else if(n%2==0){
    System.out.println("Even");
}else{
    System.out.println("Odd");
}
```

16. What is the output of the following code segment?

```
String s1 = "Bahamas";
String s2 = "BAHAMAS";
if(s1.toUpperCase() == s2)
    System.out.println("1");
if(s1.toUpperCase().equals(s2))
    System.out.println("2");
```

17. Write a java program that receives a text message from keyboard and print it out if its length is between 6-10 characters.

18. Write a java program that let the user choose his/her username and password. The program verifies whether the chosen username and password are valid. If either one of them is invalid, it notifies the user and explain the cause of invalidity. The rules for choosing valid usernames and passwords are:
    a. A username must be of length 6-15 characters.
    b. A username must start with an uppercase English alphabet A to Z.
    c. A password must not be shorter than 8 characters but must not exceed 256.
    d. There cannot be any types of parentheses or whitespaces in a valid username or password.
    e. A password cannot contain or be the same as its associated username.

Chapter 7: Iterations

Objectives

Readers should

- Be able to use Java iterative constructs, including **`do-while`**, **`while`**, and **`for`**, as well as the nested version of those constructs correctly.
- Be able to design programs that require the inclusion of iterations.

Repetitive Execution

In writing most of useful computer programs, it is necessary to have the ability to execute a set of statements repeatedly for a certain number of iterations or until some conditions are met or broken. In Java, such ability is provided through three iterative constructs, namely **`do-while`**, **`while`** and **`for`** statements.

'do-while' Statement

A **`do-while`** statement is of the form:

```
do{
    Actions
}while(Boolean Expression);
```

Its associated program flow can be shown in the following figure.

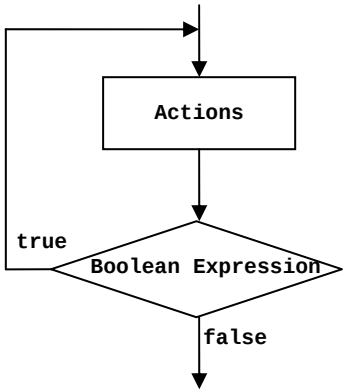


Figure 112: A flowchart representing a do-while statement

`Actions` can be one or more statements that will be repeatedly executed as long as **`Boolean Expression`** is evaluated to **`true`**. Once the program reaches the **`do-while`** statement, **`Actions`** will be execute first. Then, **`Boolean Expression`** is evaluated, and its value determines whether the program flow will loop back to repeat **`Actions`**, or finish the **`do-while`** statement.

'while' statement

Another way to execute a set of statements repeatedly until a specified condition is met is to use a **`while`** statement. A **`while`** statement is of the form:

```
while(Boolean Expression){
    Actions
}
```

Its associated program flow can be shown in the following figure.

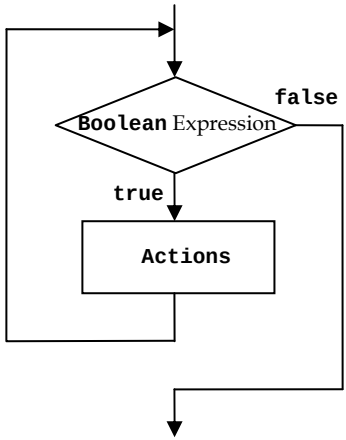


Figure 113: A flowchart representing a while statement

Actions can be one or more statements that will be repeatedly executed as in the **do-while** case. However, before the **while** block is entered, **Boolean Expression** is checked. If its value is **false**, the statements in the **while** block will not be executed. If its value is true, **Actions** will be executed, and after that, the program flow loops back to checking **Boolean Expression**.

Example 53: Basic loops with while and do-while

Observe how the **do-while** statement works by looking at the following program and its output.

```
public class DoWhileDemo1                                          1
{                                                                  2
       public static void main(String[] args)                     3
       {                                                           4
           int i = 1;                                              5
           final int N = 5;                                        6
           do{                                                     7
               System.out.println("Iteration # "+i);              8
               i++;                                                9
           }while(i<=N);                                           10
           System.out.println("Out of while loop when i="+i);      11
       }                                                           12
}                                                                  13
```

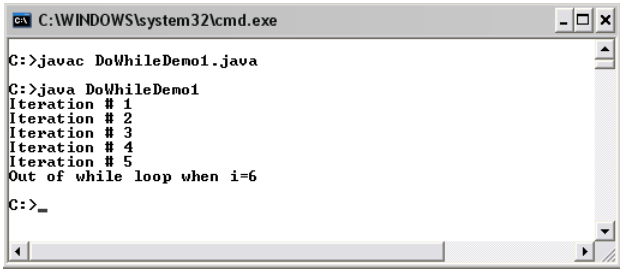


Figure 114: A program coded to run five iterations of statements using a do-while loop

Initially, this program sets **i** to 1. This variable can be thought of as a counter of how many times the statements in the **do-while** block have already been executed. Each time this **do-while** block is entered, the program prints the iteration number from the variable **i**, which is increased by 1 at the end of every iteration (on line 9) before the program checks the **boolean** value of **i<=N**, where **N** equals 5. The program will exit the **do-while** statement after the 5th iteration, at the end of which the value of **i** is 6.

The following program performs the same task as the program above but with the use of a **while** statement instead of the **do-while** one.

```
public class WhileDemo1                                              1
{                                                                    2
       public static void main(String[] args)                       3
       {                                                             4
           int i = 1;                                                5
           final int N = 5;                                          6
           while(i<=N){                                              7
               System.out.println("Iteration # "+i);                 8
               i++;                                                  9
           };                                                        10
           System.out.println("Out of while loop when i="+i);        11
       }                                                             12
}                                                                    13
```

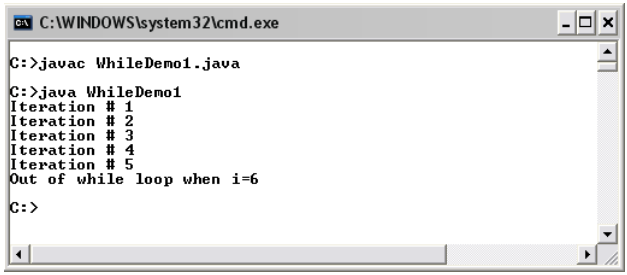


Figure 115: A program coded to run five iterations of statements using a while loop

---

Both programs in Example 53 produce the same outputs. The only difference between using a **do-while** statement and a **while** statement is that the statements in the **do-while** block is always executed at least once since the condition checking is done after the **do-while** block, while the checking is done prior to ever entering the **while** block. Thus, the statements in the **while** block may never be executed.

Example 54: q for quit

The programs in this example shows how to use keep prompting for character input from the user repeatedly until a specified character is entered. The first one does nothing much apart from waiting for a 'q' character to be entered.

```
import java.io.*;                                                    1
public class WhileMenuDemo                                          2
{                                                                    3
    public static void main(String[] args) throws IOException       4
    {                                                                5
        boolean done = false;                                        6
        char command;                                                7
        BufferedReader stdin =                                       8
            new BufferedReader(                                      9
                new InputStreamReader(System.in));                   10
        while(!done){                                                11
            System.out.print("Enter a character (q to quit): ");     12
            command = stdin.readLine().charAt(0);                    13
```

(continued on next page)

(continued from previous page)

```
            if(command == 'q') done = true;                    14
        }                                                      15
    }                                                          16
}                                                              17
```

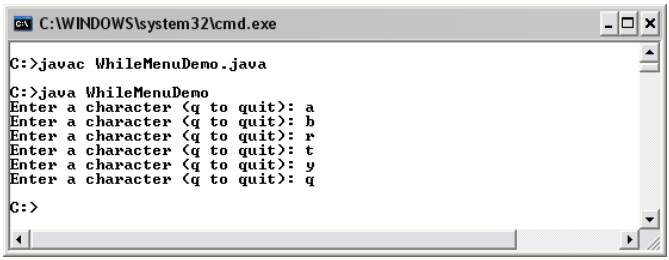


Figure 116: A program that keeps prompting for input

On line 6, a **`boolean`** variable called **`done`** is created and initialized to **`false`**. This variable is used in the condition checking of the **`while`** statement so that the statements in the **`while`** block will be iteratively executed as long as **`done`** is **`false`**. **`done`** has to be set to **`true`** at some point to avoid infinite loop (i.e. the situation when the iteration repeats forever!), and in this program, it is when the **`char`** value that the user enters equals **`'q'`**, as on line 14.

The following program finds average of a number of values entered by the user. The program iteratively asks the user to enter each value at a time, until a character 'q' is entered.

```
import java.io.*;                                              1
public class Average1                                          2
{                                                              3
    public static void main(String[] args) throws IOException  4
    {                                                          5
        double sum = 0; int i = 0;                             6
        boolean doneInputing = false;                          7
        String input;                                          8
        BufferedReader stdin =                                 9
            new BufferedReader(                                10
                new InputStreamReader(System.in));             11
        System.out.println("Please enter each value at a time."); 12
        System.out.println("Enter \'q\' when finished.");      13
```

(continued on next page)

(continued from previous page)

```
        while(!doneInputing){                                          14
            System.out.print("-- Enter value #"+(i+1)+" : ");          15
            input = stdin.readLine();                                  16
            if((input.length()==1) && (input.charAt(0)=='q')){         17
                doneInputing = true;                                   18
            }else{                                                     19
                i++;                                                   20
                sum += Double.parseDouble(input);                      21
            }                                                          22
        }                                                              23
        System.out.println("Average = "+(sum/i));                      24
    }                                                                  25
}                                                                      26
```

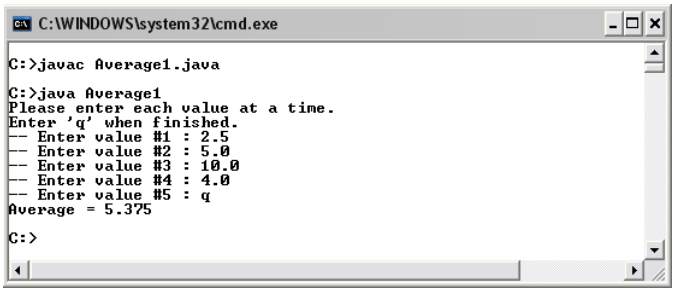


Figure 117: A program used to find the average of numbers

Try for yourself to write a rather similar program that finds the average of the values input by the user, but the program asks how many values will be entered first. Then, if the user specifies that there will be `n` values, the program iteratively prompts the user for each input `n` times before calculating the average of those values and shows the result on the screen. Use a `while` statement or a `do-while` statement.

Example 55: Guessing a number

The following program is called GuessGame.java. The user of this program will play a game in which he/she needs to guess a target number, which is a number that the program has randomly picked in the range that the user chooses. The program will repeatedly prompt for the

guessed number and provide a clue whether the guessed number is bigger or smaller than the target number, until the guessed number equals the target number.

Many things that we have learned so far are used in this program, including method calling, conditional constructs, data type casting, iterative constructs, and etc. Thus, you should observe the source code and be able to understand every statement used in this program.

```
import java.io.*;                                                    1
public class GuessGame                                               2
{                                                                    3
  public static void main(String[] args) throws IOException          4
  {                                                                  5
    int x=0, y=0, target, nTrial=0, guess;                           6
    boolean validBound = false;                                      7
    boolean notCorrect = true;                                       8
    String comparison;                                               9
                                                                     10
    BufferedReader stdin =                                           11
        new BufferedReader(new InputStreamReader(System.in));        12
    System.out.print("\nGuess an integer “);                        13
    System.out.println(“in the range of X to Y");                    14
    System.out.println("-------------------------------------");     15
                                                                     16
    while(!validBound){                                              17
        System.out.print("First, enter an integer for X : ");       18
        x = Integer.parseInt(stdin.readLine());                      19
        System.out.print("Then, enter an integer for Y : ");        20
        y = Integer.parseInt(stdin.readLine());                      21
        if(y>x){                                                     22
            validBound = true;                                       23
        }else{                                                       24
            System.out.println("-- !! Y must be greater than X.");   25
        }                                                            26
    }                                                                27
                                                                     28
    target = (int)Math.round(x+Math.random()*(y-x));                 29
                                                                     30
    System.out.println("....");                                     31
    System.out.print("A random integer from "+x+" to "+y);          32
    System.out.println(" was created.");                             33
    System.out.println("Guess it!");                                 34
    System.out.println("-------------------------------------");     35
                                                                     36
    while(notCorrect){                                               37
        nTrial++;                                                    38
        System.out.print("\nTrial #"+nTrial+"-->");                  39
```

(continued on next page)

(continued from previous page)

```
        guess = Integer.parseInt(stdin.readLine());                 40
        if(guess == target){                                        41
            notCorrect = false;                                     42
            System.out.println("-- Yes! You've got it right!");     43
        }else{                                                      44
            comparison = (guess>target)? "big.":"small.";           45
            System.out.print("-- Your guess is too ");               46
            System.out.println(comparison);                         48
        }                                                           49
    }                                                               50
                                                                    51
    System.out.println("-------------------------------------");   52
    System.out.println(nTrial+" attempts used.");                   53
                                                                    54
  }                                                                 55
}                                                                   56
```

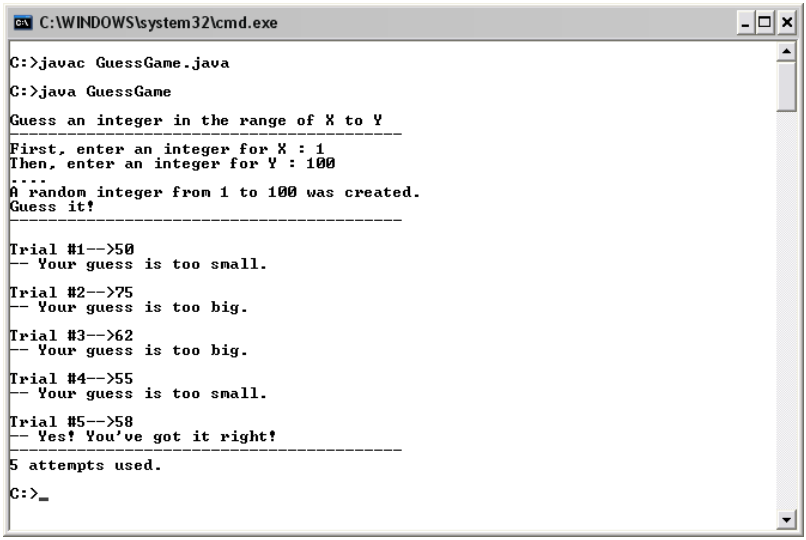


Figure 118: A guessing game program

'for' statement

Another form of an iterative construct is the **`for`** statement, which is of the form:

```
for(Init Expression; Cond Expression; Update Expression){
    Actions
}
```

which is equivalent to:

```
Init Expression;
while(cond Expression){
    Actions
    Update Expression;
}
```

Its associated program flow can be shown in the following figure.

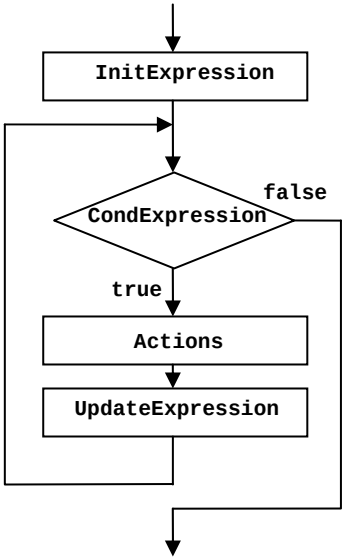


Figure 119: A flowchart representing a for statement

The process starts with the execution of **InitExpression**. This is usually for assigning an initial value to a variable, which is often of type **int**. Then, the program checks whether **CondExpression** is evaluated to **true**. If so, the program executes **Actions**. If not, the program goes out of the **for** statement. **CondExpression** usually involves checking the value of the variable initialized in **InitExpression**. Once the program finishes the execution of **Actions**, **UpdateExpression** is executed. **UpdateExpression** typically involves changing the value of the variables used in **CondExpression**. Variables determining how many times **Actions** will be executed are called *index variables*.

Here are some examples of **for** loops.

```
for(int i=1; i<=10; i++){
    System.out.println(i);
}//for loop A

for(int i=10; i>0; i--){
    System.out.println(i);
}//for loop B

for(int i=0; i<=10; i += 2){
    System.out.println(i);
}//for loop C

for(int i=1; i<100; i *= 2){
    System.out.println(i);
}//for loop D

for(char c='A'; c<='Z'; c++){
    System.out.println(c);
}//for loop E
```

for loop A prints the values of **i** from 1 to 10. **for** loop B prints the values of **i**, starting from 10 down to 1. **for** loop C prints 0, 2, 4, 6, 8, and 10. **for** loop D prints 1, 2, 4,8, 16, 32, 64. And, **for** loop E prints A to Z.

If needed, there can be more than one **Init Expression**'s as well as more than one **Update Expression**'s. Each of them is separated using commas. Look at such an example below. Notice that the initialization part contains two assignments, **i=0** and **j=10**, while the updating part contains **i++** and **j--**.

```
for(int i=0, j=10; i<=j; i++, j--){
    System.out.println(i+","+j);
}
```

The above **for** loop causes the program to print:

```
0,10
1,9
2,8
3,7
4,6
5,5
```

Example 56: Averaging input numbers

The following Java program finds the average of a number of values input by the user using a **for** loop. The number of values to be averaged is entered and stored in the variable **n** on line 11. Then, the **for** loop, starting on line 12, executes the statement located inside the loop (on line 13 and on line 14) for **n** iterations. Notice how the variable **n** and the variable **i** are used in order to iteratively execute the statements inside the loop **n** times.

```
import java.io.*;                                                    1
public class Average2                                                2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
    double sum = 0; int n = 0;                                       6
    String input;                                                    7
    BufferedReader stdin =                                           8
        new BufferedReader(new InputStreamReader(System.in));        9
    System.out.print("How many values you want to average? : ");    10
    n = Integer.parseInt(stdin.readLine());                          11
    for(int i=1;i<=n;i++){                                           12
        System.out.print("-- Enter value #"+i+" : ");               13
        sum += Double.parseDouble(stdin.readLine());                 14
    }                                                                15
    System.out.println("Average = "+(sum/n));                        16
  }                                                                  17
}                                                                    18
```

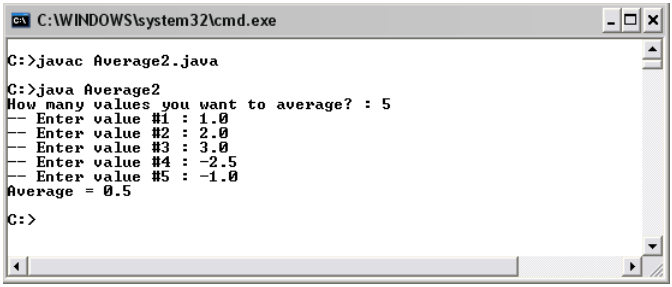


Figure 120: A program calculating the average of input data where the number of data point can be determined via keyboard

---

Example 57: Prime factors

Let's look at a program called Factorization.java which is a program that finds prime factors of an integer (e.g. the prime factorization of 120 is 2×2×2×3×5). The algorithm used here (which is by no mean the best) is that we will iteratively factor the smallest factor out. Let the integer to be factorized be **`n`**. An integer **`i`** is a factor of **`n`** when **`n%i`** equals 0. A **`for`** loop is used to perform the iteration, starting from using the index variable of 2. In each iteration of the **`for`** loop, all factors equal the index variable are factored out via a **`while`** statement. In the program, a variable **`m`** is used for storing the resulting integer of the partially factorization. The **`for`** loop continues as long as the index variable **`i`** is still less than **`m`**. We assume that the input to the program is an integer greater than 1.

```
import java.io.*;                                                    1
public class Factorization                                           2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
    int n, m;                                                        6
    BufferedReader stdin =                                           7
        new BufferedReader(new InputStreamReader(System.in));        8
    System.out.print("Enter an integer : ");                        9
    n = Integer.parseInt(stdin.readLine());                          10
    m = n;                                                           11
    for(int i=2;i<=m;i++){                                           12
        while(m%i == 0){                                             13
            System.out.print(i+", ");                                14
            m = m/i;                                                 15
```

(continued on next page)

(continued from previous page)

```
        }                                                    16
    }                                                        17
    System.out.println("\n");                               18
  }                                                          19
}                                                            20
```

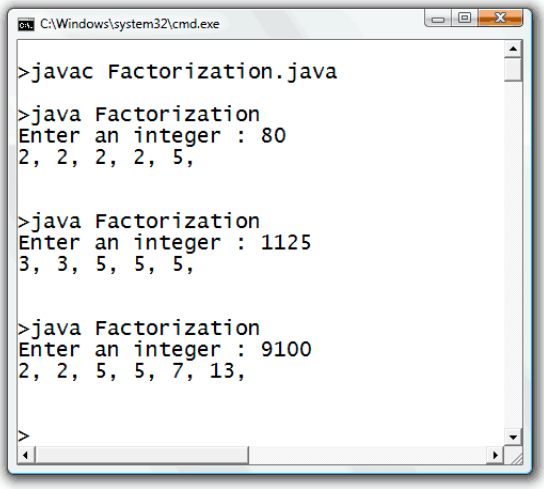


Figure 121: A program that finds prime factors of an input integer

---

Example 58: Recurrence relations

A sequence of number $\{a_n\} = a_0, a_1, a_2, a_3, \ldots$ can be defined using recurrent relation, in which the value of the n^{th} term of the sequence (a_n) is described based on preceding terms. For the first-order recurrent relation, the only preceding term required to compute the value of a_n is a_{n-1}. Thus, the first-order recurrent relation can be written as:

$$a_n = ma_{n-1} + k,$$

where m and k are two constant values defining the relation. We can compute the value of a_n in the sequence described by a first-order recurrent relation for any positive integer n from its initial condition, which is the value of a_0, and its associated recurrent relation.

The following program finds the value of a_1 to a_n for any positive integer n defined by the user from a first-order recurrent relation and its initial condition. A **for** loop is used for computing a_i from a_{i-1} for i equals 1 to the specified n.

```
import java.io.*;                                                    1
public class RecurrenceRelation                                     2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {                                                                  5
      double an, an_1, k, m, a0;                                     6
      int n;                                                         7
      BufferedReader stdin =                                         8
          new BufferedReader(new InputStreamReader(System.in));      9
                                                                     10
      System.out.print("\nRecurrence Relation:");                   11
      System.out.println(" a(n) = m*a(n-1) + k\n");                  12
                                                                     13
      System.out.print("m    --> ");                               14
      m = Double.parseDouble(stdin.readLine());                      15
      System.out.print("k    --> ");                               16
      k = Double.parseDouble(stdin.readLine());                      17
      System.out.print("a(0) --> ");                               18
      a0 = Double.parseDouble(stdin.readLine());                     19
      System.out.print("n    --> ");                               20
      n = Integer.parseInt(stdin.readLine());                        21
      System.out.println("---------------------");                  22
                                                                     23
      an_1 = a0;                                                     24
      for(int i=1;i<=n;i++){                                         25
          an = m*an_1+k;                                             26
          System.out.println("a("+i+") = "+an);                      27
          an_1 = an;                                                 28
      }                                                              29
  }                                                                  30
}                                                                    31
```

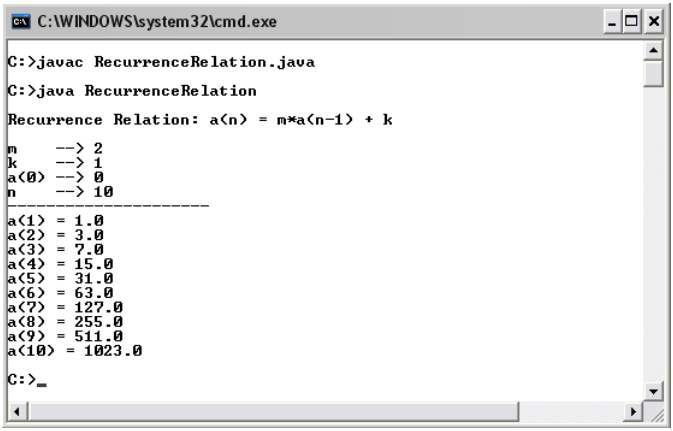


Figure 122: A program calculating term values in a first-order recurrence relation

---

'break' and 'continue'

We have seen in the previous chapter that a **`break`** statement causes the program to jump out of the current conditional construct and continue executing statements following that construct. If a **`break`** statement is put inside an iterative construct, it causes the program to jump out of that construct; no matter how many times the loop is left to be executed.

Example 59: The magic word

The following program is a trivial program that persistently prompts the user to enter some texts. It will keep prompting the user for infinitely many times unless the user enters **`Java`**.

```
import java.io.*;                                                    1
public class BreakDemo1                                              2
{                                                                    3
    public static void main(String[] args) throws IOException       4
    {   String s;                                                    5
```

(continued on next page)

(continued from previous page)

```
        BufferedReader stdin =                                          6
            new BufferedReader(new InputStreamReader(System.in));       7
        while(true){                                                    8
            System.out.print("Say the magic word\n>>");                 9
            s = stdin.readLine();                                       10
            if(s.equals("Java")) break;                                 11
        }                                                               12
        System.out.println(":)");                                       13
    }                                                                   14
}                                                                       15
```

We can see on line 8 that the condition for the **`while`** loop is always **`true`**. Thus, the **`while`** loop repeats forever unless the input *String* is "Java", in which case the **`break`** statement is executed, resulting in the program jumping out of the **`while`** loop.

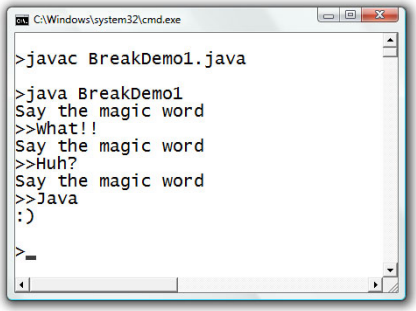


Figure 123: Demonstration of using the break statement

Figure 123 shows a screenshot when the BreakDemo1 program is executed. From the figure, we can observe that **`while(true)`** is performed three times before in the third iteration the **`break`** statement is called.

A **`continue`** statement causes the current iteration to terminate immediately. However, unlike what happens with a **`break`** statement, a **`continue`** statement pass the program flow to the start of the next iteration.

Example 60: The biggest digit in an alphanumeric String input

The following program should give you an example of how **continue** works. The program is used for finding a maximal-valued digit from a string of character. Each character in the input string is not restricted to being a digit. For example, if the input is "abc12D81", the maximal-valued digit is 8.

```
import java.io.*;                                                    1
public class ContinueDemo1                                           2
{                                                                    3
  public static void main(String[] args) throws IOException          4
  {     int len, max = 0;                                            5
        String s;                                                    6
        BufferedReader stdin =                                       7
            new BufferedReader(new InputStreamReader(System.in));    8
        System.out.print("Enter any string with digits : ");        9
        s = stdin.readLine();                                        10
        len = s.length();                                            11
        for(int i=0; i<len; i++){                                    12
            char c = s.charAt(i);                                    13
            if(!(c>='0' && c<='9')) continue;                        14
            int digit = Character.digit(c,10);                       15
            if(digit > max) max = digit;                             16
        }                                                            17
        System.out.println("Max digit --> "+max);                    18
  }                                                                  19
}                                                                    20
```

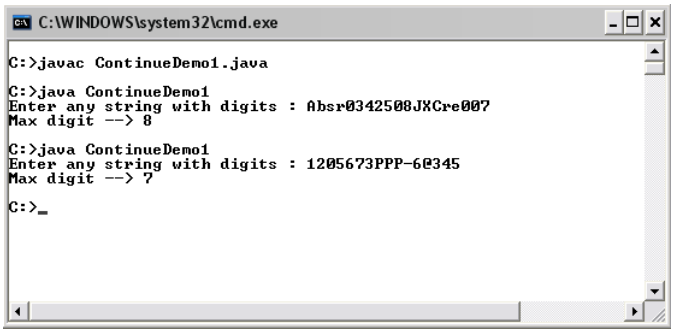


Figure 124: Demonstration of using the continue statement which allows the program to ignore statements in some certain iteration

A **for** statement starting on line 12 goes through every position of the input *String*. In each position, if the character at that position does not fall between '0' and '9' inclusively, that character is not a digit. Thus, if **!(c>='0' && c<='9')** is **true**, there is no need to do anything else to the character in that position. Therefore, **continue** is used in order for the program to start the next iteration right away, as seen on line 14.

Note that the expression **Character.digit(c,10)** returns the numeric value of **c** in the decimal (base 10) system.

Nested Loops

An iterative construct can be placed inside another iterative construct causing what we called *nested loops*. Figuring out the program flow of nested loops does not require any extra knowledge apart from what we have discussed so far. We can just think about the inner loop as an operation that needs to be finished (kept iterating until the loop terminating condition is met) for each iteration of the outer loop.

The following code segment is an example of a **for** loop placed inside another **for** loop. Note that **Actions** is to be replaced with a set of statements.

```
for(int i=1; i<=n; i++){
    for(int j=1; j<=m; j++){
            Actions
    }
}
```

The outer **for** statement is associated with an index variable **i** that runs from 1 to **n**, resulting in **n** iterations of the inner **for** statement. For each iteration of the outer **for** statement, the inner **for** statement iterates **m** times. So, this results in **Actions** being executed **n** × **m** times.

The following code segment, using nested **while** constructs performs similar actions to the one above.

```
int i=1;
while(i<=n){
    int j=1;
    while(j<=m){
            Actions
            j++;
    }
    i++;
}
```

Example 61: Nested loops

The program listed below use nested **for** loops in which the inner loop runs for a fixed number of iteration regardless of the value of the index variable used in the outer loop.

```
public class NestedLoopDemo1                                    1
{                                                               2
        public static void main(String[] args)                 3
        {                                                       4
            for(int x=1; x<=3; x++){                            5
                for(char y='A'; y<='C'; y++){                   6
                  System.out.println(x+"-"+y);                  7
                }                                               8
            }                                                   9
        }                                                       10
}                                                               11
```

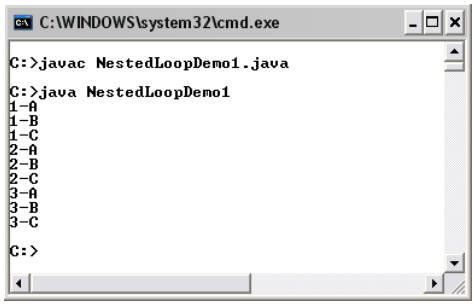


Figure 125: An example of nested loops in action

In this program, the outer **for** loop starts by setting the index variable **i** to 1. For each iteration of this loop, the statements in the inner **for** loop is

performed three times, with the inner loop's index variable **y** being '**A**', '**B**', and '**C**'.

The nested loops used in the following program are different than the ones used above in the aspect that the number of iteration of the inner loop depends on the index variable of the outer loop.

```
public class NestedLoopDemo2                                    1
{                                                               2
       public static void main(String[] args)                  3
       {                                                        4
           for(int x=1; x<10; x++){                             5
               System.out.println("x="+x);                      6
               System.out.print("  --> y=");                    7
               for(int y=1; y<=x;y++){                          8
                 System.out.print(y+",");                       9
               }                                                10
               System.out.print("\n");                        11
           }                                                    12
       }                                                        13
}                                                               14
```

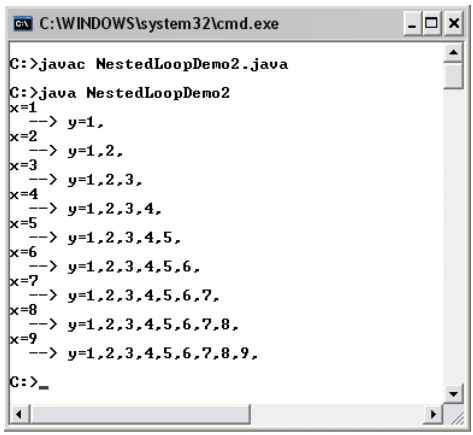


Figure 126: Another example of nested loops in action

This example shows nested **for** loops in the case that the number of iteration of the inner loop depends on a variable that changes with each iteration of the outer loop. As we can see from the screenshot in Figure

126, the upper bound of the values of **y** is **x** which changes with each iteration of the outer loop from 1 to 9.

Scope of Variables

If index variables are declared in the initialization part of a **for** statement (i.e. in **InitExpression**), they are known only inside the block associated with that **for** loop. Or, we can say that the scope of these variables is just inside that **for** statement. If you attempt to use these variables outside, a compilation error will occur.

In contrary, a variable declared in a block associated with a **for** loop can be used in any for loops nested inside that block and locating after the variable declaration. The same rule applies to any kind of blocks in a Java program. This includes blocks associated with **do-while**, **while**, **if**, **if-else**, **if-else-if**, **switch**, and etc.

Example 62: Variable scope

The following code segment cannot be compiled successfully since the variable **k** is not known outside the **for** loop.

```
public static void main(String[] args)
{       int lastIndex;
        for(int k = 0; k < 10; k++){
            // Some statements
        }
        lastIndex = k; // This line causes a compilation error.
}
```

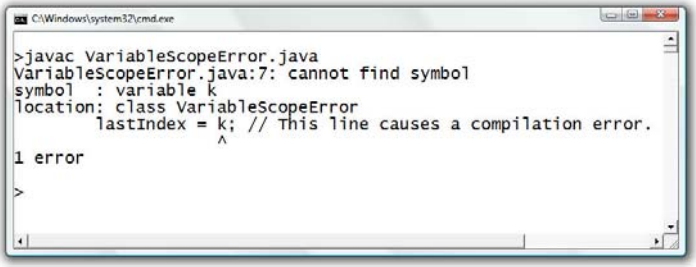


Figure 127: A compilation error caused by that a variable being declared inside a for loop is used outside

Consider another program below.

```
public class VariableAndBlock1 {                                    1
    public static void main(String [] args){                        2
        int x;                                                       3
        if(true){                                                    4
            x = 1;                                                   5
            int y = x;                                               6
            while(y<10){                                             7
                y++;                                                 8
            }                                                        9
        }                                                            10
        System.out.println(y);                                       11
    }                                                                12
}                                                                    13
```

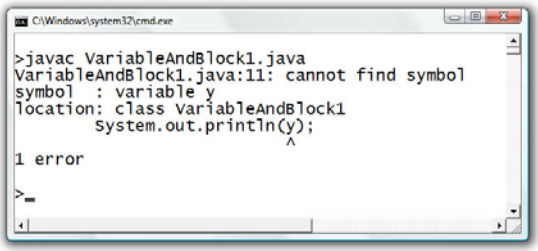


Figure 128: Another example of error caused by a variable's scope

The program cannot be compiled due to an error about the variable `y` is not known outside the `if` statement in which it is declare. Readers should notice that the compiler only complains about the use of `y` on line

11. It does not complain anything about the usage of **y** on line 7 and line 8 since both lines are inside a block nested inside the **if** statement where **y** is declared and the lines come after the variable's declaration.

Let's conclude this chapter with an interesting example that can help solving real mathematic problems.

Example 63: All solutions to $a^2+b^2+c^2=200$

(x,y,z) is a solution of $a^2+b^2+c^2 = 200$ if the equation is true when a = x, b = y, and c = z. How many unique solutions are there if we require that a, b, and c can only take non-negative integers? We can write a Java program utilizing nested **for** loops to count the number of all possible solutions to the equation that meet the non-negative integer requirement.

The idea is to try all possible combinations of three non-negative integers and see whether which ones of them satisfy $a^2+b^2+c^2 = 200$. Due to the non-negative integer requirement, each variable from a, b, and c can only be as small as 0, while each of them cannot exceed $\lfloor\sqrt{200}\rfloor$. Thus, we use three-level nested *for* loops, each of which is associated with a variable that runs from 0 to $\lfloor\sqrt{200}\rfloor$. Whether each combination of the three non-negative integers is a solution of the equation is tested in the innermost *for* loop, where the solution is also printed out on the screen.

Here is a Java program that does the mentioned task.

```
public class SolutionsCount                                          1
{                                                                    2
  public static void main(String[] args)                            3
  { int a,b,c,nSolutions=0;                                          4
    int maxPossible = (int)Math.sqrt(200);                           5
    for(a = 0; a < maxPossible; a++){                                6
      for(b = 0; b < maxPossible; b++){                              7
        for(c = 0;c < maxPossible; c++){                             8
          if(a*a+b*b+c*c == 200){                                    9
            nSolutions++;                                            10
            System.out.print("Soltn # "+nSolutions+" is ");         11
            System.out.println("("+a+","+b+","+c+")");                12
          }                                                          13
```

(continued on next page)

(continued from previous page)

```
        }                                                                    14
      }                                                                      15
    }                                                                        16
    System.out.print("# of non-negative integer solutions for ");           17
    System.out.println("a^2+b^2+c^2= 200 is "+nSolutions);                  18
  }                                                                          19
}                                                                            20
```

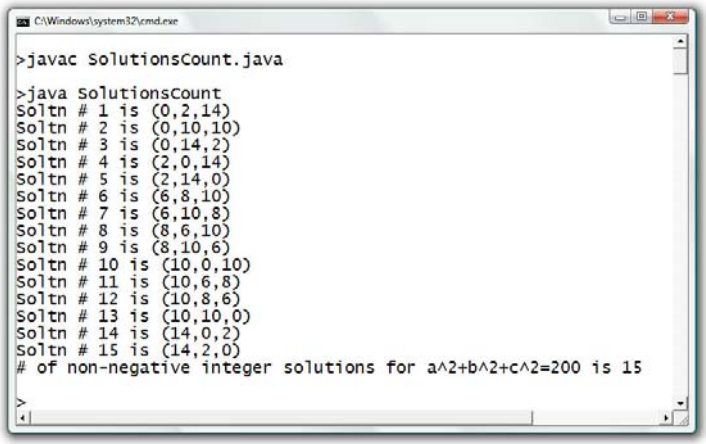


Figure 129: A program finding all possible solutions to $a^2+b^2+c^2=200$

Exercise

1. If `n` is an integer greater than 0, how many times is *boohoo()* executed in the following code segment?

```
int i = 0;
do{
    i++;
    booHoo();
}while(i<=n);
```

2. What is the output of the following code segment?

```
int k = 0, m = 6;
while(k<=3 && m>=4){
    System.out.println(k+","+m);
    k++;
    m--;
}
```

3. What is the value of `n` after the following code segment is executed?

```
int n = 1, i = 100;
while(n++<i--);
```

4. What are the values of `n` and `m` after the following code segment is executed?

```
int n=0, m=0, i=0,maxItt=300;
while(i++<maxItt) n++;
i=0;
while(++i<maxItt) m++;
```

5. Rewrite the code segment in the last problem by using *for* loops instead of the two *while* statements.

6. What is the value of `n` after the following code segment is executed?

```
int n = 0;
for(int i = 1;i<=100;i++)
    for(int j = 1;j<=100;j++)
        n++;
```

7. What is the value of `n` after the following code segment is executed?

```
int n = 0;
for(int i = 50;i>0;i--)
    for(int j = 40;j>0;j--)
        n++;
```

8. What is the output of the following code segment?

```
int k=0;
for(int i=100;i>1;i/=2,k++);
System.out.println(k);
```

9. Explain why the following program cannot be compiled successfully.

```
public class Ex7Test
{
    public static void main(String[] args)
    {
        int n = 10, k = 0;
        for(int i=0;i<n;i++){
            k++;
        }
        System.out.println("i="+i+", k="+k);
    }
}
```

10. Use a **for** statement to write a Java program to calculate the value of 8!

11. Repeat the previous problem using a **while** statement instead of the **for** statement.

12. Write a Java program that calculates *n*! from the integer *n* obtained from keyboard. If the value of the input *n* causes an overflow to occur, report it to the user. Otherwise, show the value of *n*! on screen.

13. Write a Java program that randomly picks an English alphabet (A-Z) and keeps asking the user to guess the alphabet until he/she has got it right. Also report the number of trials.

14. Determine the output of the following code segment.

```
int i,j;
for(i=0,j=0;i<=100;i++){
    if(i%2==0)
        i++;
    else
        j++;
}
System.out.println(j);
```

15. Determine the output of the following code segment.

```
int i=0,a=0,n=0;
while((i=a+=2)<=100){
    n++;
};
System.out.println(i+","+a+","+n);
```

16. What is the value of **k** after the following code segment is executed?

```
int k=0, j=0;
while(true){
    if(j<20){
        k++; j++;
    }else{
        break;
    }
    if(k>5){
        continue;
    }else{
        j++;
    }
}
```

17. Use nested loops to write a Java program that generates a multiplication table as shown below.

```
        |   2   3   4   5   6   7   8   9
        |
2       |   4   6   8   10  12  14  16  18
3       |   6   9   12  15  18  21  24  27
4       |   8   12  16  20  24  28  32  36
5       |   10  15  20  25  30  35  40  45
6       |   12  18  24  30  36  42  48  54
7       |   14  21  28  35  42  49  56  63
8       |   16  24  32  40  48  56  64  72
9       |   18  27  36  45  54  63  72  81
```

18. Write a Java program that reverses the order of the characters in the string input from keyboard and show them on screen.

19. Write a Java program that finds the value of $f(x,n)$, where x can be any real value, n can only be a non-negative integer, and $f(x,n)$ is defined as:

$$f(x,n) = \sum_{i=0}^{n} x^i$$

    Your program must check whether the value of n input by the user is valid or not. (i.e. n must have an integer value and be non-negative.) Use *Math.pow()* to find the value of x^i.

20. Repeat the previous problem without using any methods provided by the *Math* class.

21. Write a Java program that calculates and shows the sum of all even integers from 0 to **n**, where **n** is specified by the user via keyboard. Assume that **n** is an integer greater than 0.

22. An imaginary webmail service called "veryhotmail.com" is kind enough to let its users choose their own passwords. However, a valid password must comply with the following (somewhat strange) rules:

    1. The length must be between 6 to 15 characters, inclusive.
    2. Each character must be either one of the 26 English alphabets. Both uppercase and lower case letters are allowed.
    3. The difference between the number of the uppercase letters and the lowercase letters cannot be greater than 20% of the password length.

    Write a code segment that can validate the password stored in a *String* reference **s**.

23. Write a Java program to shows the value of a_0, a_1, …, a_n that is corresponding to the recurrence relation $a_k = k^2a_{k-1}-a_{k-2}+3^k$ where k = 2, 3, 4 ,… The values of n and the initial conditions (a_0 and a_1) are specified by the user at run-time via keyboard.

24. Write a Java program to find the number of solutions to the equation $x+y+z = 30$ where x, y, and z are non-negative integers.

25. Write a Java program to find the number of solutions to the equation $x+y+z = n$ where n is a constant integer supplied by the user of the program and x, y, and z are integers between $-b$ and b. Also, b is an integer defined by the user.

26. Write a Java program that receives an English sentence from keyboard. The program encodes the sentence by reversing the order of letters in each word appearing in that sentence while keeping the original word order and shows the result on screen. Assuming that words are always separated by a single space and there cannot be spaces at the beginning and the end of the sentence. For example, "**`We are the champions.`**" is encoded as "**`eW era eht .snoipmahc`**".

27. Write a code segment that replaces any number of multiple spaces connected together in a *String* reference **`s`** with single spaces.

    For example, if **`s`** contains:

    "**`This  does  not contain     multiple      spaces.`**",

    it should be changed to:

    "**`This does not contain multiple spaces.`**"

28. An integer *n* is prime if its only factors are 1 and itself. Write a code segment for checking whether the value of an **`int`** *n* is prime. Assume that *n* is a positive integer greater than 1.

Chapter 8: Methods

Objectives

Readers should

- Be able to define new methods and use them correctly.
- Understand the process of method invocation.

Methods

Sometimes it is cumbersome to write, debug or try to understand a very long program. Many times, there are some parts of the program that redundantly perform similar tasks. Actually writing statements to perform those similar tasks many times is obviously not very efficient, when one can possibly write those statements once and reuse them later when similar functionalities are needed. Programmers usually write statements to be reused in sub-programs or subroutines. This does not only allow efficient programming but also makes programs shorter which, consequently, make the programs easier to be debug and understood. Dividing statements intended to perform different tasks into different subroutines is also preferred.

Methods in Java can serve the purpose just mentioned.

The following example shows a situation in which a program should be re-written using methods.

Example 64: Introducing methods

The following program computes:

$$y = f(x,n) = x + 2x^2 + 3x^3 + ... + nx^n$$

for x = 1.5, 2.5, 3.5, and 4.5, where n = 3.

```
public class MethodDemo1                                    1
{                                                           2
        public static void main(String[] args)             3
        {                                                   4
            double x1,x2,x3,x4;                             5
            double y;                                       6
            x1 = 1.5;                                       7
            y = 0;                                          8
            for(int i=1;i<=3;i++){                          9
                y += i*Math.pow(x1,i);                      10
            }                                               11
            System.out.println(y);                          12
            x2 = 2.5;                                       13
            y = 0;                                          14
            for(int i=1;i<=3;i++){                          15
                y += i*Math.pow(x2,i);                      16
            }                                               17
            System.out.println(y);                          18
            x3 = 3.5;                                       19
            y = 0;                                          20
            for(int i=1;i<=3;i++){                          21
                y += i*Math.pow(x3,i);                      22
            }                                               23
            System.out.println(y);                          24
            x4 = 4.5;                                       25
            y = 0;                                          26
            for(int i=1;i<=3;i++){                          27
                y += i*Math.pow(x4,i);                      28
            }                                               29
            System.out.println(y);                          30
        }                                                   31
}                                                           32
```

We can observe that for each value of **x**, a **for** statement is used for computing **y**. Writing the program this way makes the source code longer than it is necessary. Furthermore, if the programmer made a logical mistake in implementing the calculation of *f(x,n)* (which is not the case in this example), the mistake would have to be fixed in many places. Instead, we can make the functionality for computing *f(x,n)* a method and call the method once for each value of **x**. This can result in the following code.

```
public class MethodDemo2                                    1
{                                                           2
        public static void main(String[] args)             3
        {                                                   4
            double x1,x2,x3,x4;                             5
```

(continued on next page)

(continued from previous page)

```
            x1 = 1.5;                                                6
            System.out.println(f(x1,3));                             7
            x2 = 2.5;                                                8
            System.out.println(f(x2,3));                             9
            x3 = 3.5;                                                10
            System.out.println(f(x3,3));                             11
            x4 = 4.5;                                                12
            System.out.println(f(x4,3));                             13
        }                                                            14
                                                                     15
        public static double f(double x,int n){                     16
            double y = 0;                                            17
            for(int i=1;i<=n;i++){                                   18
                y += i*Math.pow(x,i);                                19
            }                                                        20
            return y;                                                21
        }                                                            22
}                                                                    23
```

From the code, observe that the program is not composed of only the method *main()* like every program that we have seen so far anymore. Instead, on line 16 to line 22, we can see a segment of code whose structure looks a lot like the method *main()*. This segment of code is called the definition of method *f()*, which is defined and given its name in this program. This method is responsible for carrying out the iterative computation of **y** based on the value of **x** and **n**. On line 7, line 9, line 11, and line 13, this method is used to compute the value of **y** based on **x** and **n** put in the parentheses of *f()* on each line.

Do not panic yet. At this point you are only expected to adopt a rough idea of how one can define methods and make use of them. Next, we will look at the detail structure (syntax) of how to use and define a method.

Using a Method

Using a method should not be new to you. Consider the following statement.

```
double y = Math.pow(2.0,3.0);
```

We should know by now that the statement computes 2^3 and assigns the resulting **double** value to **y**. What we should pay attention to now is the fact that the method takes a **double** as its first argument, and another **double** as its other argument. The **double** value of the first argument is raised to the power of the value of the second **double** argument. The resulting value, which we assign to **y** in the above statement, is said to be *returned* from the method.

Observe from the use of method that has already be defined in a standard Java class, we can see that to define a new method we at least have to define what its argument list, how the arguments should be used, and what the method should return.

Defining a Method

The definition of a method is of the form:

```
public static returnType methodName(
                          argType1 arg1,
                          argType2 arg2,
                          …,
                          argTypeN argN)
{
    methodBody
}
```

Here is an example of what the form above could look like. This example shows how *Math.pow()* could be defined in side the *Math* class.

```
public static double pow(double a, double b)
{

    /* some statements that perform
       the calculation and return the
       result to the caller */

}
```

The first two words, **public** and **static**, are Java keywords. **public** identifies that this method can be used by any classes. **static** identifies

that this method is a class method. Now we will just use these two keywords as they are.

`returnType` should be replaced with the name of the data type expected to be returned by the method. **`returnType`** can be any of the eight primitive data types or the name of a class. When a method returns something, only one value can be returned. When a method does not return any value, a keyword **`void`** is used for **`returnType`**.

`methodName` should be replaced with the name (identifier) you give to the method. Java naming rules apply to the naming of methods as well as other identifiers.

Inside the parentheses is the argument list consisting of parameters expected to be input to the method. Each parameter in the list is declared by identifying its type (any of the eight primitive data types or the name of a class) followed by the name (identifier) to be used in the method for that parameter. When the method does not need any input parameters, do not put anything in the parentheses.

`methodBody` is the list of statements to be executed once the method is called. These statements might be referred to as the body of the method. The body of the method might contain a **`return`** statement, in which the keyword **`return`** is placed in front of the value wished to be returned to the caller of the method. You need to make sure that the value returned is of the same type as, or can be automatically converted to, what is declared as **`returnType`** in the method header. The **`return`** statements also mark terminating points of the method. Whenever a **`return`** statement is reached, the program flow is passed from the method back to the caller of the method. If there is nothing to be returned, i.e. **`returnType`** is **`void`**, the keyword **`return`** cannot be followed by any value. In this case, the program flow is still passed back to the caller but there is no returned value.

Now let's look again at the *f()* method in the previous example. Matching line 16 of the previous example with the form we have just mentioned, we see that the method returns a **`double`**. Its argument list consists of a **`double`** and an **`int`**. **`x`** is used as the name of the **`double`** parameter, while **`n`**

is used as the name of the other parameter. Line 17 to line 21 is the body of the method. The keyword **return** is used to identify the value that will be returned by the method to its caller. In this example, **return y;** indicates that the value to be returned is the value of the variable **y**.

Here are some examples of method definitions.

```
public static boolean isOdd(int n){
    return (n%2 != 0)? true : false;
}
```

```
public static int unicodeOf(char c){
    return (int)c;
}
```

```
public static String longer(String s1, String s2){
    return ((s1.length() > s2.length())?s1:s2);
}
```

```
public static int factorial(int n){
    int nFact = 1;
    for(int i=1;i<=n;i++){
        nFact *= i;
    }
    return nFact;
}
```

```
public static boolean hasSimilarChar(String s1, String s2){
    boolean similarChar = false;
    for(int i=0; i<s1.length() && !similarChar; i++){
        for(int j=0; j<s2.length(); j++){
            if(s1.charAt(i) == s2.charAt(j)){
                similarChar = true;
                break;
            }
        }
    }
    return similarChar;
}
```

```
public static void printGreetings(String name){
    System.out.println("Hello "+name);
    System.out.println("Welcome to ISE mail system.");
    System.out.println("--------------------------");
}
```

```
public static double h(double a, double b, double c, double d){
    double num = g(a);
    double den = g(a)+g(b)+g(c)+g(d);
    return num/den;
}
```

```
public static double g(double d){
    return Math.exp(-d/2);
}
```

Notice that, in the last example, two methods, *h()* and *g()*, are defined. In the body of *h()*, *g()* is called several times. This shows that you can call methods that you define by yourself inside another method definition in a similar fashion to when you call them from *main()*.

Example 65: Angle between two vectors

Let's look at an example of writing a Java program where tasks are performed via several separate methods. Here, we wish to find the angle θ between two vectors in the Cartesian coordinate, both started at (0,0), as depicted in the figure below.

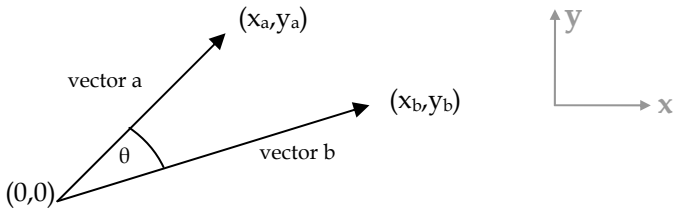


Figure 130: Two vectors in the Cartesian coordinate

Problem definition: The program needs to calculate the angle θ between two vectors in the Cartesian coordinate supplied by the user.

Analysis: Inputs are the two vectors. Since both of them always start at (0,0), the program only need to know the coordinate of the ending point of each vector. Coordinates can be specified using two numeric values representing location in the *x* and *y* directions. Therefore, to specify input vectors, two pairs of (*x*,*y*) should be input by entering each value at a time via the keyboard. Calculation of the angle based on the input should then be done and shown on screen.

The resulting angle should be in degrees and shown nicely on screen using the maximum of two decimal points.

Design:

- Prompt the user to input the required coordinates: x_a, y_a, x_b, and y_b. Store them in **`double`** variables.

- Calculate θ from:

$$\theta = \cos^{-1}\left(\frac{x_a x_b + y_a y_b}{\sqrt{x_a^{\,2} + y_a^{\,2}}\sqrt{x_b^{\,2} + y_b^{\,2}}}\right)$$

- $\sqrt{x_a^{\,2} + y_a^{\,2}}$ and $\sqrt{x_b^{\,2} + y_b^{\,2}}$ are very similar and each of them corresponds to the length of its associated vector. Thus, we could consider finding each square root as calculating vector length. Then, the resulting lengths are multiplied together and used as the denominator of the above formula.

- θ calculated from the above formula is in radian. It needs to be converted to degree using *Math.toDegree()*.

- Show the resulting angle in degree on screen.

The above step can be depicted as the program flow below.

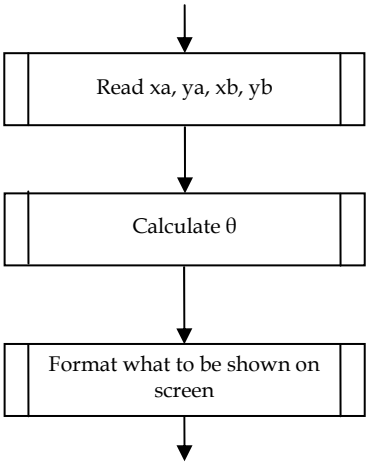


Figure 131: A flowchart showing various subroutines used in the VectorAngle program

Implementation: An implementation of the program could be similar to the code shown below. Notice that the instructions of the program seen in *main()* are divided into three major subroutines as described in Figure 131. Furthermore, most significant functionalities inside each of these subroutines are also achieved by calling methods, some of which are available via existing Java classes such as *java.io.BufferedReader* while some are newly defined in our program.

```
import java.io.*;                                                    1
public class VectorAngle                                             2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {   double xa,ya,xb,yb,theta;                                      5
      xa = readDouble("xa = ");                                      6
      ya = readDouble("ya = ");                                      7
      xb = readDouble("xb = ");                                      8
      yb = readDouble("yb = ");                                      9
      theta = calculateAngle(xa,ya,xb,yb);                           10
      System.out.print("Angle between ("+xa+","+ya+") and");        11
      System.out.print(" ("+xb+","+yb+") is ");                      12
      System.out.println(setDecimalPoints(theta,2)+" degrees.");     13
  }                                                                  14
```

(continued on next page)

(continued from previous page)

```
  public static double readDouble(String s) throws IOException    15
  {   BufferedReader stdin =                                       16
          new BufferedReader(new InputStreamReader(System.in));   17
      System.out.print(s);                                         18
      return Double.parseDouble(stdin.readLine());                 19
  }                                                                20
  public static double                                             21
  calculateAngle(double x1, double y1, double x2, double y2)       22
  {   double len1, len2, thetaInRad;                               23
      len1 = length(x1,y1);                                        24
      len2 = length(x2,y2);                                        25
      thetaInRad = Math.acos((x1*x2+y1*y2)/(len1*len2));           26
      return Math.toDegrees(thetaInRad);                           27
  }                                                                28
  public static double length(double a,double b)                   29
  {                                                                30
      return Math.sqrt(a*a+b*b);                                   31
  }                                                                32
  public static String setDecimalPoints(double d,int n)           33
  {                                                                34
      double nPowerOf10 = Math.pow(10,n);                          35
        String nDecimalPoints =                                    36
          "."+(int)(Math.round(d*nPowerOf10)%nPowerOf10);          37
        String integerDigit = (int)d+"";                           38
        return integerDigit+nDecimalPoints;                        39
  }                                                                40
}                                                                  41
```

Readers are encouraged to analyze the statements in *setDecimalPoints()*, which is the method that creates a *String* representation of the input number with the specified number of digits in the fraction part of that number, and see how they work.

Testing: A screenshot of the program is shown in Figure 132. In reality, a number of test cases where the resulting angles are known should be input to the program in order for the tester to manually check the consistency of the results. In this figure, you can observe informal attempts in testing the program in several conditions, including the conditions where the results are expected to be 0, 90, and 180 degrees.

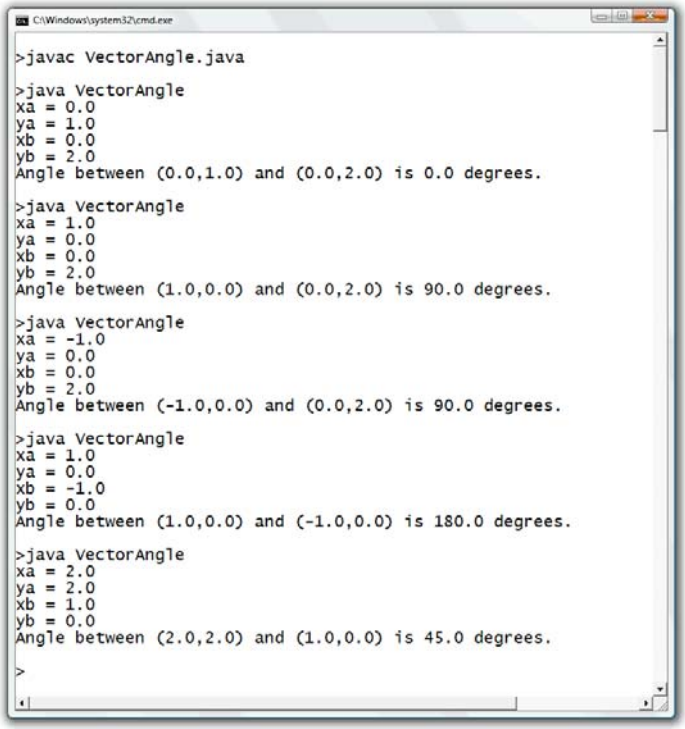


Figure 132: A program finding the angle between two vectors

Multiple Return Statements

It is possible for a method to contain multiple return statements but all of them have to return values of the type declared in the method header and each **`return`** statement must be reachable.

The following method implements the function whose definition is:

$$f(x)=\begin{cases} x+1 & ;0\le x<1 \\ x^2 & ;1\le x<2 \\ 0 & otherwise \end{cases}$$

```
public static double f(double x){
    if(x>=0 && x<1){
        return x+1;
    }else if(x>=1 && x<2){
        return x*x;
    }else{
        return 0;
    }
}
```

Notice the method contains 3 **return** statements. They are all reachable but which one is executed depending on the value of **x**. When the execution of the method reaches any one of the **return** statements, the program flow is returned to the caller along with the associated returned value right away.

Now consider a trivial method *g()* as listed below. Compiling a source code with such a method gives compilation errors due to the fact that under no circumstances that the second and third **return** statement will be executed.

```
public static double g(){
    return 1.0;
    return 2.0;
    return 3.0;
}
```

Local Variables

Variables declared in the argument list of the method header are available throughout the method body, but not outside of the method. The variables are created once the method is entered and destroyed once the method is terminated.

Variables declared inside the method body are available inside the block they are declared as well as blocks nested in the block they are declared.

Within the block, variables are available after they are declared. Also, they are destroyed once the method is terminated.

Example 66: Method's local variables

The following program yields a compilation error due to a missing variable.

```
public class ScopeError                                                 1
{                                                                       2
       public static void main(String[] args)                          3
       {                                                                4
           int x=0, y=0, z;                                             5
           z = f(x,y);                                                  6
           System.out.println("myMultiplier = "+myMultiplier);          7
           System.out.println("z="+z);                                  8
       }                                                                9
       public static int f(int a, int b)                                10
       {   int myMultiplier = 256;                                      11
           return myMultiplier*(a+b);                                   12
       }                                                                13
}                                                                       14
```

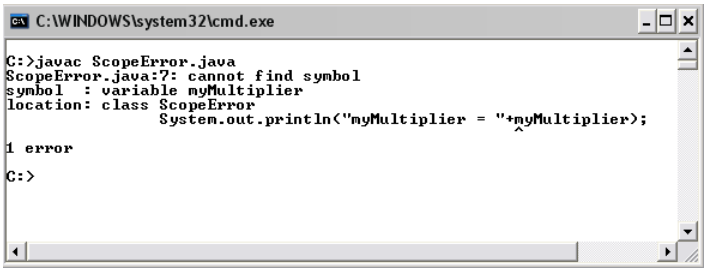


Figure 133: Compilation error due to usage of a variable declared inside a method outside the method

Method Invocation Mechanism

When a method is invoked, typical steps involving the values passed between the caller and the method as well as the method's local variables are:

1. If there are input arguments to the method, the value of each argument is copied to its corresponding local variable declared inside the method header.

2. Local variables are declared and assigned with values according to the statements implemented in the method.

3. If the method returns a value back to the caller, the returned value is copied from inside the method to the caller. From the scope of the caller, the expression corresponding to the invoked method is evaluated to that returned value.

4. When the program flow is returned to the caller, all local variables of the method are no longer accessible by the program.

Example 67: Parameter passing and returning values

Consider the following program, which calculates $(6^2+8^2)^{1/2}$ using the method *f()* which can calculate $(a^n+b^n)^{1/n}$ for any numeric values a and b, and any integer value n. We will learn about the mechanism that takes place when a method is called from this example.

```
public class MethodInvokeDemo                                    1
{                                                                2
       public static void main(String[] args)                   3
       {                                                         4
           double x = 6.0, y = 8.0, z;                           5
           z = f(x,y,2);                                         6
           System.out.println(z);                                7
       }                                                         8
       public static double f(double a,double b, int n)         9
       {                                                         10
           double an = Math.pow(a,n);                            11
           double bn = Math.pow(b,n);                            12
           return Math.pow(an+bn,1.0/n);                         13
       }                                                         14
}                                                                15
```

When the method is called (`z=f(x,y,2);`), the following steps take place.

1. The program flow is passed to *f()* whose definition starts on line 9. Variables in the input argument list of the method header (line 9) are created. In this case, two variables of type **`double`** are created and named **`a`** and **`b`**, while another **`int`** variable is created and named **`n`**.

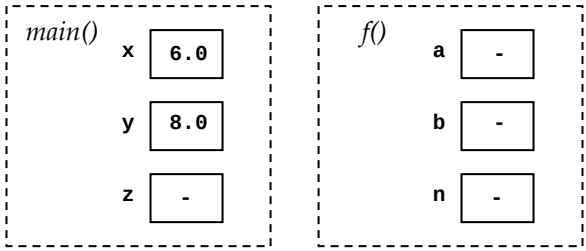


Figure 134: Illustration of variables when f() is invoked(Step1)

2. Each variable is assigned with its appropriate value. By calling **`f(x,y,2)`**, **`a`** is assigned with the value of **`x`**, **`b`** is assigned with the value of **`y`**, and **`n`** is assigned with 2. Note that **`x`** and **`a`** do not share the same memory location, the value of **`x`** is just copied to a once. It is the same for **`y`** and **`b`**.

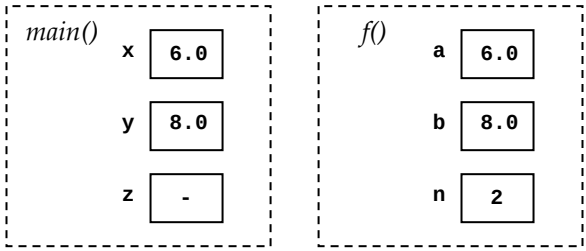


Figure 135: Illustration of variables when f() is invoked (Step2)

3. Then, the statements in the method body are executed. Line 11 and line 12 caused **`an`** and **`bn`** to be created and assigned values. Remember that both variables are only local to *f()*. Before

returning the value to *main()*, `Math.pow(an+bn,1.0/n)` is evaluated to 10.0.

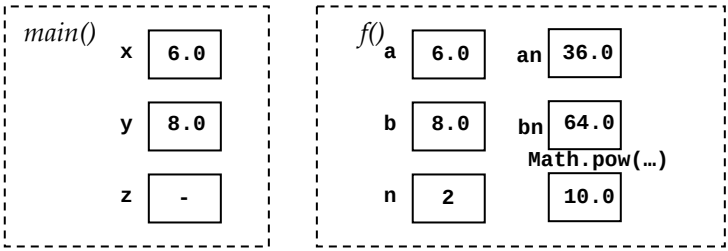


Figure 136: Illustration of variables when f() is invoked (Step3)

4. The value of `Math.pow(an+bn,1.0/n)` is copied to the variable `z` in *main()* as the result of the assignment operator. Variables local to *f()* are then destroyed and the program flow is passed back to *main()*.

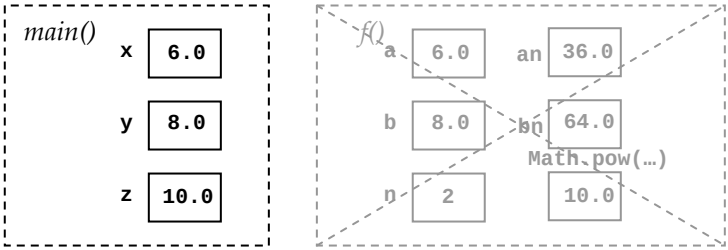


Figure 137: Illustration of variables when f() is invoked (Step4)

It is important to keep in mind that variables local to *main()* as well as variables local to different methods are available inside the methods that they are declared. Thus, it is possible, and is usually the case that, variable names are reused in different methods. For example, the variables `a` and `b` in *f()* could be named as `x` and `y` without any confusion.

The fact that variables are local to the method they are declared enables programmers to re-use variable names as long as the scopes of variables of similar names are not overlapping.

Example 68: Different variables with the same identifiers

Observe the following program and its output.

```
public class MethodVariableName                                    1
{                                                                  2
       public static void main(String[] args)                     3
       {                                                           4
           int x=1,y=1,w;                                          5
           w = add(x,y);                                           6
           System.out.println("x="+x+"y="+y);                      7
       }                                                           8
       public static int add(int x,int y)                          9
       {                                                           10
           int z = x+y;                                            11
           x = 0;                                                  12
           y = 0;                                                  13
           return z;                                               14
       }                                                           15
}                                                                  16
```

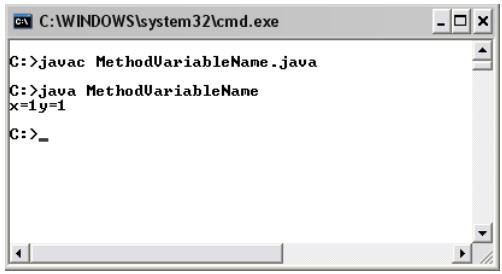


Figure 138: A program demonstrating variable name re-use

If you are not surprised with the output (`x=1 y=1`), that is good. You may skip this paragraph. If you think that the output should be `x=0 y=0`, you should note that `x` and `y` declared in *main()* on line 5 are different from `x` and `y` declared in *add()* on line 9. Thus, changing that value of `x` and `y` local to *add()* does not have anything to do with `x` and `y` local to *main()*, which are the ones printed out on screen.

Example 69: Value swapping

This example aims at discussing the difference between two programs. The first program is called SwapDemo in which the values of two variables are swapped with each other. A common way to swap values between two variables is to introduce another temporary variable and use it to store the original value of one variable before it is overwritten with the value of the other variable. Statements on line 6 to line 8 of the following program perform the swapping of the values of two variables. The screenshot of the program's output in Figure 139 shows that it works as we want.

```
public class SwapDemo                                                    1
{                                                                        2
        public static void main(String[] args)                          3
        {   int a=9, b=8, temp;                                          4
            System.out.println("a="+a+" b="+b);                         5
            temp = a;                                                    6
            a = b;                                                       7
            b = temp;                                                    8
            System.out.println("Swapped!\na="+a+" b="+b);               9
        }                                                                10
}                                                                        11
```

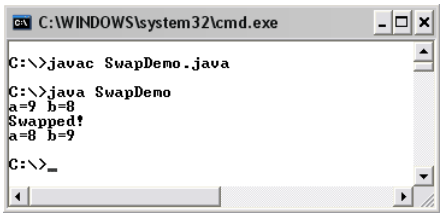


Figure 139: A program that swaps values of two variables

Now, let's say that we would like to write this swap functionality as a method so that whenever such a swapping is needed, the method can then be invoked conveniently. The new program could end up like the following.

```
public class SwapDemoWrong                                               1
{                                                                        2
```

(continued on next page)

(continued from previous page)

```
        public static void main(String[] args)                    3
        {   int a=9, b=8, temp;                                   4
            System.out.println("a="+a+" b="+b);                   5
            swap(a,b);                                            6
            System.out.println("Swapped!\na="+a+" b="+b);         7
        }                                                         8
        public static void swap(int a, int b){                   9
            int temp;                                             10
            temp = a;                                             11
            a = b;                                                12
            b = temp;                                             13
        }                                                         14
}                                                                 15
```

Statements on line 6 to line 8 of the SwapDemo program are re-implemented inside *swap()* in this new program. The *swap()* method takes two input arguments whose values should be swapped by the method. However, the output of the program shown in Figure 140 suggests that there are something wrong with this method implementation.

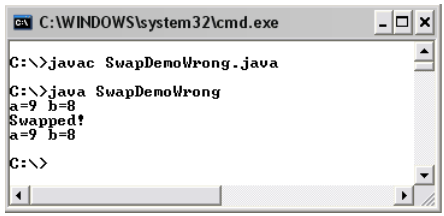


Figure 140: An incorrect implementation of a method that swaps values of its two input argument

What's wrong is that the only values that are swapped are the values of **a** and **b** local to the *swap()* method. Note that they are not the values of **a** and **b** that are local to the *main()* method. As we can see from the output of the program, the values of **a** and **b** of the *main()* method are still intact. In the case that you are still confused, try following the steps described in the "Method Invocation Mechanism" section.

Unfortunately, there is no easy way to implement a method that really swaps the values of its input argument if the types of the input arguments are primitive data types.

Passing by Value Vs. Passing by Reference

When passing input arguments to methods, Java copied values of the arguments into their corresponding local variables in the methods. Utilizing this way of passing arguments, it is said that Java pass the arguments by values. Therefore, changes made to the method's local variables are unrelated to the values of the original variables. This *pass-by-value* mechanism applies to all variables used as inputs to any Java methods regardless of whether they are primitive or non-primitive.

Some programming languages (such as C, C++) support another way of passing argument in which changes made to local variables declared in the methods (or some other kinds of subroutine) affect the values of their respective original variables that are supplied as input arguments to the methods. This way of passing argument is referred to as a *pass-by-reference* mechanism.

Method Overloading

Different methods, even though they behave differently, can have the same name as long as their argument lists are different. This is called *Method overloading*. Method overloading is useful when we need methods that perform similar tasks but with different argument lists, i.e. argument lists with different numbers or types of parameters. Java knows which method to be called by comparing the number and types of input parameters with the argument list of each method definition. Readers should realize that methods are overloaded based on the difference in the argument list, but not based on their return types.

Example 70: Overloaded number adding

Consider the following program.

```
public class OverloadingDemo                                    1
{                                                               2
  public static void main(String[] args)                       3
  {                                                             4
```

(continued on next page)

(continued from previous page)

```
      System.out.println(numericAdd(1,2));                          5
      System.out.println(numericAdd(1,2,3));                        6
      System.out.println(numericAdd(1.5,2.5));                      7
      System.out.println(numericAdd(1.5,2.5,3.5));                  8
      System.out.println(numericAdd('1','2'));                      9
  }                                                                 10
                                                                    11
  public static int numericAdd(int x,int y)                         12
  {                                                                 13
      return x + y;                                                 14
  }                                                                 15
  public static double numericAdd(double x,double y)                16
  {                                                                 17
      return x + y;                                                 18
  }                                                                 19
  public static int numericAdd(int x,int y, int z)                  20
  {                                                                 21
      return x + y + z;                                             22
  }                                                                 23
  public static double numericAdd(double x,double y, double z)     24
  {                                                                 25
      return x + y + z;                                             26
  }                                                                 27
  public static int numericAdd(char x, char y)                      28
  {   int xInt = x - '0';                                           29
      int yInt = y - '0';                                           30
      return xInt+yInt;                                             31
  }                                                                 32
}                                                                   33
```

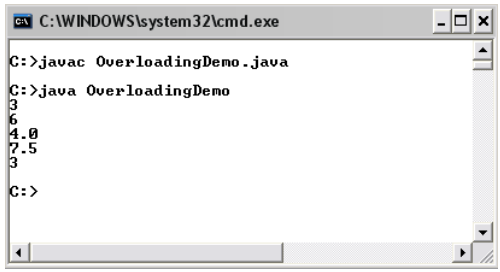


Figure 141: A program demonstrating method overloading

In this program, we overload methods *numericAdd()*. On line 5 to line 9, we call *numericAdd()* with different argument lists. Based on the input parameters, methods with appropriate definition are activated.

`numericAdd(1,2)` activates `numericAdd(int x,int y)`.

`numericAdd(1,2,3)` activates `numericAdd(int x,int y,int z)`.

`numericAdd(1.5,2.5)` activates `numericAdd(double x,double y)`.

`numericAdd(1.5,2.5,3.5)` activates `numericAdd(double x,double y,double z)`.

Finally, `numericAdd('1','2')` activates `numericAdd(char x,char y)`.

Now, let's look at some examples of incorrect method overloading.

```
public static int f(int x, int y){
    …
}
public static double f(int x, int y){
    …
}
```

Both methods are not counted as methods overloading since the method names as well as their argument lists are the same. Regardless of their return types, they are considered the same methods. Consequently, their definitions are considered redundant, and, therefore, cause a compilation error.

```
public static int g(int x, int y){
    …
}
public static int g(int a, int b){
    …
}
```

The *g()* methods in the this example differ only in the variable names. This difference does not make the two argument lists different. The numbers and types of parameters are the same. We only call each parameter differently, and this does not count as method overloading. Therefore, their definitions are considered redundant. Again, this causes a compilation error.

Example 71: How overloaded methods are selected

Observe the following program. Pay attention to the way Java selects which method to be called.

```
public class OverloadingDemo2                                    1
{                                                                2
  public static void main(String[] args)                        3
  {                                                              4
      h(1,1);                                                    5
      h(1.0,1.0);                                                6
      h(1,1.0);                                                  7
  }                                                              8
                                                                 9
  public static void h(int x,int y)                              10
  {                                                              11
      System.out.println("h(int x, int y) is called.");         12
  }                                                              13
  public static void h(double x,double y)                        14
  {                                                              15
      System.out.println("h(double x, double y) is called.");   16
  }                                                              17
}                                                                18
```

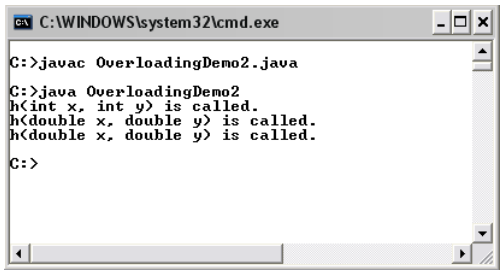


Figure 142: A program demonstrating method overloading including a case where the input argument list is not entirely matched with any overloaded methods

`h(1,1)` on line 5 is clearly corresponding to `h(int x,int y)` due to its argument list. Similarly, the method `h(1.0,1.0)` on line 6 is clearly corresponding to `h(double x,double y)`. One interesting point is which method corresponds with `h(1,1.0)`. There is no exact match for the argument list `(int,double)`. However, `int` can be automatically converted to double via widening data type conversion. Therefore, the

first input parameter in `h(1,1.0)`, which is an `int` 1, is converted to 1.0 first. Then, `h(double x, double y)` is called.

The following program cannot be compiled successfully since `g(1.0,1.0)`does not match any overloaded methods, and none of its parameters can be converted so that it matches any overloaded methods.

```
public class OverloadingDemo3
{
    public static void main(String[] args)
    {
        g(1.0,1.0);
    }
    public static void g(int x,double y)
    {
        System.out.println("g(int x, double y) is called.");
    }
    public static void g(double x,int y)
    {
        System.out.println("g(double x, int y) is called.");
    }
}
```

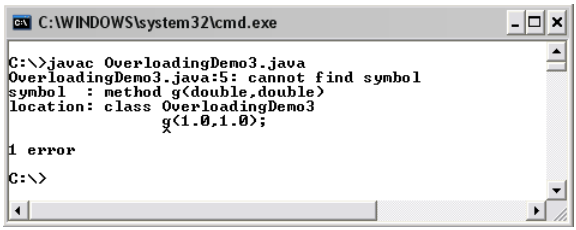


Figure 143: Compilation error due to unmatched overloaded methods

---

Exercise

1. Explain the benefits of having a program perform some sets of instruction inside methods. Can you think of any downsides of doing so?

2. Write a method header for the method *m()* each item that makes the execution of the statement in that item valid.

```
a.  int i = m(1,1);
b.  float f = m(Math.exp(5));
c.  String s = m(2f,8d);
d.  IseStudent l = m("John","K.","Maddy");
e.  for(double d=1;d<=256;d *= 2) m(d);
```

3. What is the output when *main()* is run?

```
public static void main(String[] args)
{
    System.out.println(g("A"));
}
public static String f(){
    System.out.println("A");
    return "A";
}
public static String g(String s){
    return f()+s;
}
```

4. Explain why the following code segment cannot be compiled successfully.

```
public static void main(String[] args)
{
    int i = f(2,3);
}
public static int f(int a, int b){
    return Math.pow(a,b)+Math.pow(b,a);
}
```

5. Can the following program be compiled successfully? If so, what is the output?

```
public class Ex8_5
{
    public static void main(String[] args){
        f(5);
        System.out.println("k="+k);
    }
    public static void f(int n){
        int k = 0;
        for(int i=0;i<n-1;i++){
            k += k*i;
        }
    }
}
```

6. Determine the output of the following code segment.

```
public static void main(String[] args)
{   int n = 0;
    for(int i=0;i<5;i++){
        n = f(n++);
    }
    System.out.println(n);
}
public static int f(int n)
{
    return n++;
}
```

7. Determine the output of the following code segment.

```
public static void main(String[] args)
{   int n = 0;
    for(int i=0;i<5;i++){
        n = f(++n);
    }
    System.out.println(n);
}
public static int f(int n)
{
    return ++n;
}
```

8. Given the definition of *a()* as the following:

```
public static double a(double d)
{
    return 3*d+1;
}
```

   Use *a()* to find the value of k_n for n=4,
   where $k_n = 3k_{n-1}+1$ and $k_0 = 0$;

9. Write a method called *cube()* that returns its **`double`** parameter raised to the third power.

10. Write a method called *blankLine()* used for inserting an empty line to a message. Therefore, **`"First line\nSecond line"+blankLine()+"Third line."`** would appear as:

```
First line
Second line

Third line.
```

11. Write a method called *readDigitString()* that returns a new *String* whose character sequence consisted of the input provided by the user via keyboard if the input is a valid digit string of any length. Otherwise, the method returns `null`. The method takes a *String* as its input parameter. The method prompts the user for the keyboard input by showing the message in that *String*.

12. Tax rates in a specific country can be calculated from an individual's income (in G.) obtained during the past tax year according to the following table.

Income (G.)	Tax Rate (%)
1-100,000	0
100,001-500,000	10
500,001-1,000,000	20
1,000,001-4,000,000	30
above 4,000,000	37

    According to the table, if a person's income is 550,000 G., there is no tax for the first 100,000 G., 10% of the income in the range 100,001-500,000 are taxed, which equals 40,000 G., The last 50,000 G. is taxed with the rate of 20% resulting 10,000 G. Therefore, the total tax for this person is 60,000 G.

    Write a method called *tax()* which returns the amount of tax associated with the income supplied as the only input to the method. Assume that there are no decimal points in any incomes.

13. Write a method called *nBits()* that calculates the minimum number of bits (in integer) required for using binary code to represent n different symbols. n is input to the method as the only input argument.

14. The chi-square distribution function $f(x;k)$ is defined as:

$$f(x;k) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{(k/2)-1} e^{-x/2}$$

when $x \geq 0$, k is an integer greater than 0, and $f(x;k)$ is zero when $x < 0$.

Write a method for finding the value of the chi-square distribution function at the input values of x and k. Assume that the value of $\Gamma(a)$ can be obtained by calling a class method called *gamma()* of a class called *MyMath* and use a as the only input.

15. Determine the output of the following program.

```
public class Ex8_15
{
    public static void f()
    {
        System.out.println("A");
    }
    public static void f(int a, int b)
    {
        System.out.println("B");
    }
    public static void f(float a, float b)
    {
        System.out.println("C");
    }
    public static void f(double a, double b)
    {
        System.out.println("D");
    }
    public static void f(char a, char b)
    {
        System.out.println("E");
    }

    public static void main(String[] args)
    {
        f();
        f(1,2);
        f(1.0,2.0);
        f(1,2.0);
        f(1F,2.0);
        f('1','2');
        f('1',2);
    }
}
```

16. Write overloaded methods named *nextValue()*.

    If the input is numeric value of the type int, float, or double, the associated methods should return a value that is one

greater than the input parameter but with the data type similar to the input.

If the input is a single character either a char or a *String*, the associated methods should return a char or a *String* whose value is the character immediately following the input parameter. For example, **`nextValue('q')`** must return '**`r`**'.

If the input is a *String* with more than one characters, the method should return a new *String* whose content is the same as the input but the last position is the character immediately following the character in the last position of the original *String*. For example, **`nextValue("ABC")`** must return "**`ABD`**".

If the input is **`null`**, the method returns **`null`**.

17. Write a Java program that displays positive integer values less than 10 million in words. The values are received from keyboard. For example, the value 1500 is displayed as "**`one thousand five hundred`**". The program should keep asking for another value until the user input -1.

Chapter 9: Arrays

Objectives

Readers should

- Be able to define, initialize, and use one-dimensional as well as multidimensional arrays correctly.
- Be able to use arrays as well as their elements as parameters to methods.
- Be able to write code to sort array elements in any orders desired.
- Be able to write code to search for elements in an array.
- Be able to use arrays in problem solving using computer programs.

Requirement for a List of Values

Suppose we want to write a program that counts the number (frequency) of each digit from 0 to 9 in a *String* entered by the user, the program might be written using what we have studied so far be like this:

```
import java.io.*;
public class CountDigitFrequency
{
       public static void main(String[] args) throws IOException
       {   BufferedReader stdin
               = new BufferedReader(
                   new InputStreamReader(System.in));
           System.out.print("Enter string:");
           String s = stdin.readLine();
           int freq0 = 0; int freq1 = 0;
           int freq2 = 0; int freq3 = 0;
           int freq4 = 0; int freq5 = 0;
           int freq6 = 0; int freq7 = 0;
           int freq8 = 0; int freq9 = 0;
           for(int i=0;i<s.length();i++){
               char c = s.charAt(i);
               if(c >= '0' && c <= '9'){
                   switch(c){
                     case '0': freq0++; break;
                     case '1': freq1++; break;
                     case '2': freq2++; break;
```

(continued on next page)

(continued from previous page)

```
                    case '3': freq3++; break;
                    case '4': freq4++; break;
                    case '5': freq5++; break;
                    case '6': freq6++; break;
                    case '7': freq7++; break;
                    case '8': freq8++; break;
                    case '9': freq9++; break;
                    default:
                  }
              }
          }
          System.out.println("Number of 0 = "+freq0);
          System.out.println("Number of 1 = "+freq1);
          System.out.println("Number of 2 = "+freq2);
          System.out.println("Number of 3 = "+freq3);
          System.out.println("Number of 4 = "+freq4);
          System.out.println("Number of 5 = "+freq5);
          System.out.println("Number of 6 = "+freq6);
          System.out.println("Number of 7 = "+freq7);
          System.out.println("Number of 8 = "+freq8);
          System.out.println("Number of 9 = "+freq9);
      }
}
```

From the code, you should be able to notice that ten variables are used for storing the frequencies of the digits. Also, separate instances of *System.out.println()* are called for printing the resulting frequencies. Although the names of the variables storing the frequencies are rather similar, they are independent from one another. Therefore, we cannot make use of iterative constructs to traverse the values of these variables. Being able to do so would significantly reduce the size of the code. Not being able to use iterative constructs in this case also leads to clumsy and error-prone source code.

What we require is a mechanism that enables us to store a list of related values in a single structure, which can be referred to using a single identifier. Java provides this mechanism through the use of *arrays*.

Specifically to the above example, we need a list of ten elements, in which each element is used for storing the frequency of each digit.

One-dimensional Array

An array variable is used for storing a list of element. Similar to a variable of other data types, an array variable needs to be declared. The declaration of an array takes a rather similar syntax to the declaration of a variable of other data type, which is:

```
ElementType [] a;
```

ElementType is the data type of each element in the array. The square bracket **[]** identifies that this is an array declaration. **a** is the identifier used for referring to this array. Note that identifier naming rules applies here also.

Below are some examples when arrays of various data types are declared.

```
int [] freq;
int [] numICE, numADME;
double [] scores;
char [] charList;
String [] studentNames;
```

Note that more than one arrays of the same type can be declared in a single statement such as the one shown in the second example. Also, it is perfectly correct and common to create an array of non-primitive data type such as an array of *String* shown in the last example above.

To initialize an array, the keyword *new* is used in the following syntax.

```
a = new ElementType[n];
```

n is a non-negative integer specifying the length of **a**. Here are some examples of array initialization associated with the arrays declared in the above example.

```
freq = new int[10];
numICE = new int[5];
int n = 5;
numADME = new int[n];
scores = new double[50];
charList = new char[2*n];
studentNames = new String[40];
```

Notice that we can use any expression that is evaluated to an **int** value, as well as an explicit **int** value, to specify the length.

Declaration and initialization can be done in the same statement. For example,

```
int [] freq = new int[10];
int n = 5;
int [] numADME = new int[n];
```

Whenever an array is initialized using the keyword **new**, every of its element is set to a numeric zero if the elements are of numeric types, to **false** if the elements are of **boolean** type, and to **null** (a valid reference that refers to nothing) if the elements are of non-primitive data types.

When an array is created, the data type of its element has to be specified strictly. Only values consistent with the data typed specified can be used to fill array's slots. All elements in an array must be of the same type.

Just like variables storing values of non-primitive data types (such as *String*), array variables are reference variables. The value that is actually stored in an array variable is a reference to the real array object locating somewhere in the memory.

Figure 144 shows an illustration of what happens in the computer's memory when an array variable is created and assigned with a new array.

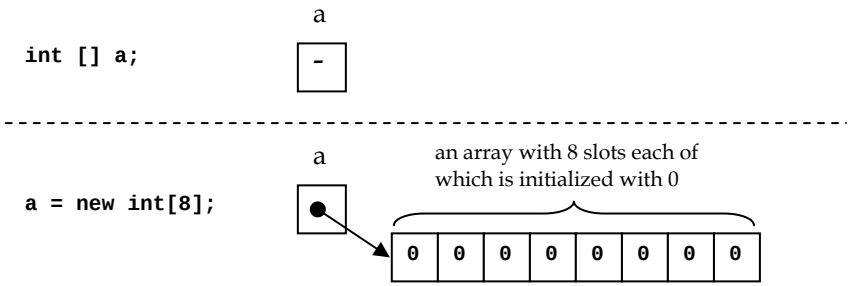


Figure 144: Illustration of memory allocation when a variable is declared and assigned with an array

Accessing Array Elements

Elements in an array can be referred to using its associated *index*. For an array of length *n*, the corresponding indexes run from 0 to n-1. An element of an array **a** with index *k* can be referred to in the program using **a[k]**.

Consider Figure 145 which shows a case when the position indexed by 3 of an array of **double** is assign with a value which is later assigned to another **double** variable.

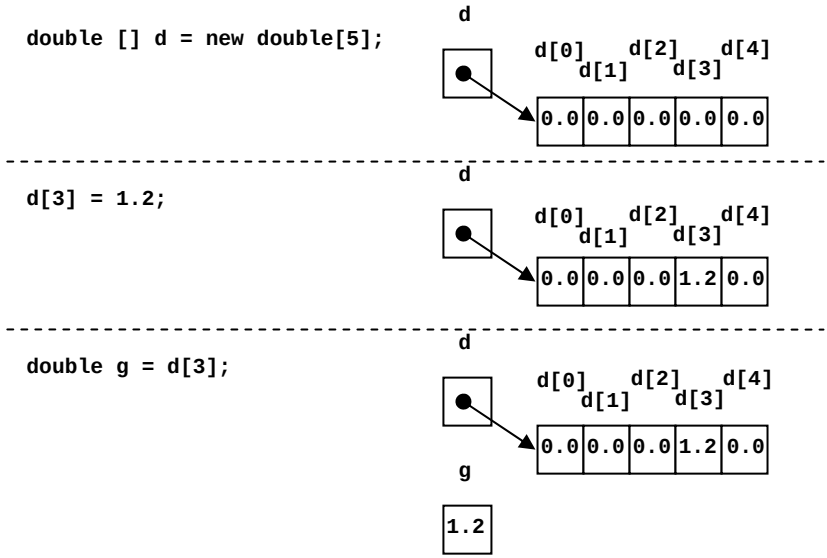


Figure 145: Illustration of accessing an array element

Example 72: Accessing array elements

Consider the following program. Note that the length of an array **a** can be found using **a.length**.

```
public class ArrayDemo1                                                 1
{                                                                       2
        public static void main(String[] args)                         3
        {                                                               4
            int [] a = new int[5];                                      5
            for(int i=0; i<a.length; i++){                              6
                System.out.print(a[i]+"\t");                            7
            }                                                           8
            System.out.print("\n");                                     9
            a[0] = 9;                                                   10
            a[1] = 10;                                                  11
            a[2] = a[0]+a[1];                                           12
            a[3] = 20;                                                  13
            a[4] = a[3]++;                                              14
            for(int i=0; i<a.length; i++){                              15
                System.out.print(a[i]+"\t");                            16
            }                                                           17
        }                                                               18
}                                                                       19
```

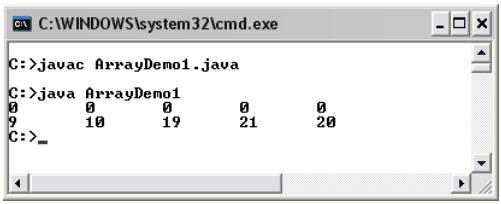


Figure 146: Demonstration of how array elements are accessed

An array of **int** of length 5 is declared and initialized on line 5. Then, a **for** loop is used to iterate through **a[i]** from **i** equals 0 to **a.length-1**. We can see that the program prints 0 0 0 0 0 out on the screen. This supports what mentioned earlier about numeric array elements being set to zeroes by default. Each element of the array **a** is assigned with different **int** value from line 10 to line 14. The **for** loop on line 15 makes the program prints the values of every element, which are 9, 10, 19, 21, and 20, on screen. Recall that the post fix increment operator on line 14 makes assign the value of **a[3]** to **a[4]** prior to increasing **a[3]** by 1.

Example 73: Improved digit frequency counting

Here we can modify CountDigitFrequency.java by storing the frequency of each digit occurrence in a single array. Appropriate iterative constructs are also used. Notice that the code is much more compact.

```
import java.io.*;                                                    1
public class CountDigitFrequency2                                    2
{                                                                    3
        public static void main(String[] args) throws IOException   4
        {   BufferedReader stdin                                     5
                = new BufferedReader(                                6
                    new InputStreamReader(System.in));               7
            System.out.print("Enter string:");                       8
            String s = stdin.readLine();                             9
            int [] freq = new int[10];                               10
            for(int i=0;i<s.length();i++){                           11
                char c = s.charAt(i);                                12
                if(c >= '0' && c <= '9'){                            13
                  freq[c-'0']++;                                     14
                }                                                    15
            }                                                        16
            for(int i=0;i<freq.length;i++){                          17
                System.out.println("Number of "+i+" = "+freq[i]);    18
            }                                                        19
        }                                                            20
}                                                                    21
```

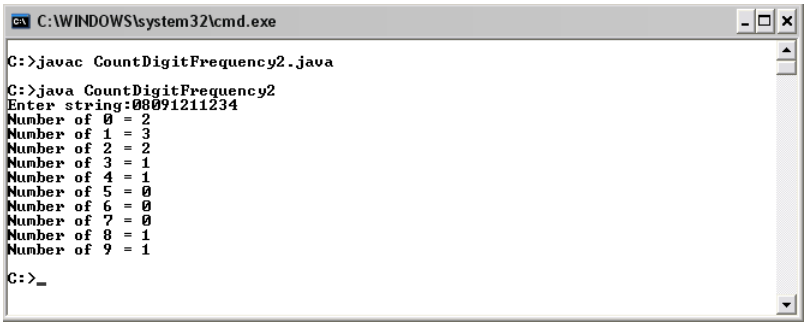


Figure 147: An improved version of a program counting digits

Instead of storing the frequencies of the ten different digits using ten separate variables, in this program, an array **`freq`** with 10 slots is used. If the character **`c`** at a position of interest falls between '0' to '9' inclusively,

the statement **freq[c-'0']++** is used for increasing the (**c-'0'**)th slot by one. Readers should recall that the value of the expression **c-'0'** is an **int** value which equals to the difference between the Unicode of the character stored in **c** and the character **'0'** (E.g. it is 0 for **'0'**, 1 for **'1'**, 2 for **'2'**, and so on).

Explicit Initialization

If we would like each element in an array to have the value other than the default value (zero for numeric types and false for **boolean**) during its initialization, we can use an *initializer list*. An initializer list is a list of values, each of which is separated by a comma, enclosed in a pair curly bracket. For example:

```
int [] a = {1,2,3,4,5};
String [] directions = {"NORTH", "EAST", "SOUTH", "WEST"};
boolean [] allTrue = {true, true, true};
```

Initializer lists have to be used in the statements in which array variables are declared. Therefore, the following statements are invalid.

```
int [] a;
a = {1,2,3,4,5};  // This is invalid.
```

Figure 148 illustrates a case when an array of **int** is initialized an initializer list and an array of *String* is initialized with another one. Also, notice the fact that values of primitive data types are stored in the array's slots directly while values of non-primitive data types are referred to by the array's slot. This works in the same way as when these data types are assigned to individual variables.

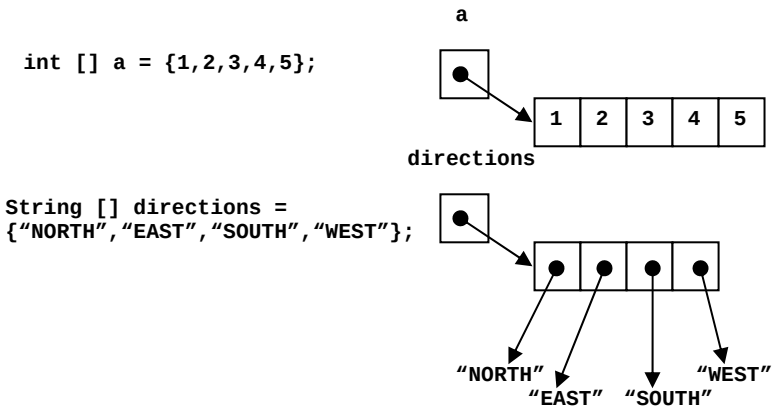


Figure 148: Illustration of some usage of initializer lists

Array Variable Assignment

Since an array variable is a reference variable, when it is re-assigned with a new array, the array originally referred to by that variable is de-referenced (and eventually destroyed). The variable then refers to the new array. Figure 149 show an example of such a case.

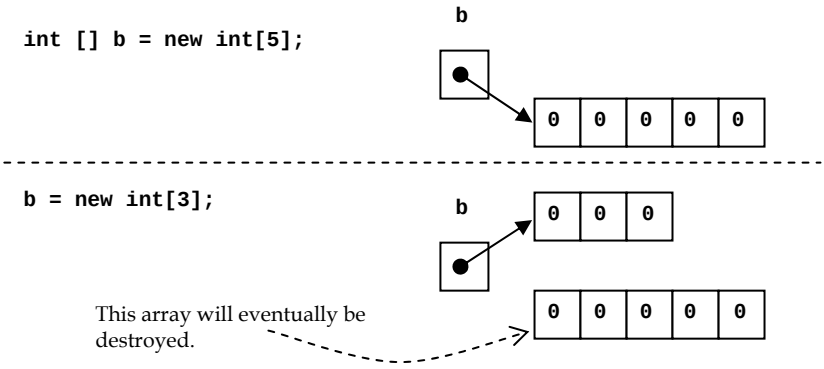


Figure 149: Assigning a new array to a variable

An array variable can be assigned with the value of another array variable of the same type. For example, the following code segment is valid.

```
char [] a = {'A','B','C'};
char [] b = {'X','Y'};
b = a;
```

Both **a** and **b** are variables referring to arrays of **char**. On the last line, **b** is made to point to the same array as **a** using the assignment operator. Notice that the fact that **a** and **b** used to refer to an array of **char** with different lengths does not matter as long as they are both arrays of the same data type.

The following code segment is invalid due to the conflicting array data types.

```
int [] k = {1,2};
double [] j = {1.0,2.0,3.0};
j = k;
```

Assigning an array variable with another array variable makes the former array variable refer to the same array as the later one. Consider Figure 150. The statement **a=b**; makes **a** refers to the same array as **b**. Therefore, **b[1]** is actually the same as **a[1]**.

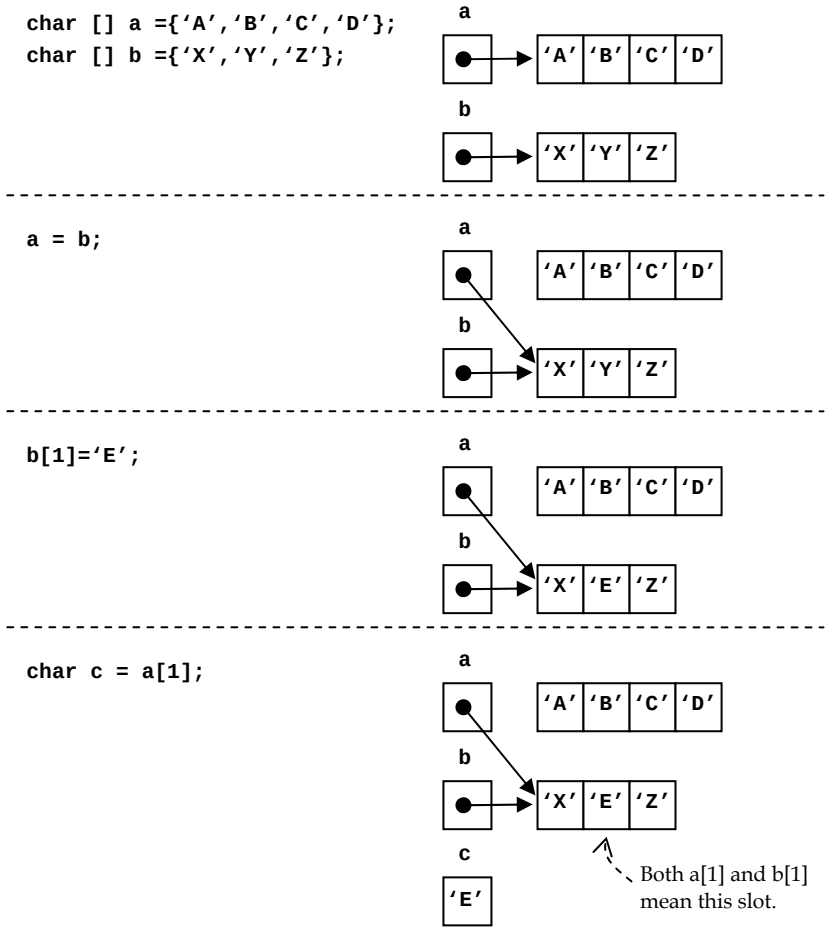


Figure 150: Assigning an array variable to another array variable

Array Utilities

Java comes with a class called *Arrays* (note the capital 'A' and the 's' suffix) that provides useful utility methods for manipulating arrays. This class is in the *java.util* package. Therefore, *java.util.Arrays* needs to be

imported in order to use the class. Many static methods relating to different kinds of array manipulation are provided so that programmers can utilize those methods by invoking them from the class name.

Here, we will only introduce some methods in the *Arrays* class that could be useful in examples presented later in this book. Theoretically, functionalities provided by some of these methods seem straightforward to implement by ourselves. However, it is also a good idea to learn to use what already exist. Readers should notice that these methods are usually overloaded so that they work for all cases of primitive data types as well as non-primitive ones.

More thorough information about this class can be found from the Java API documentation on the official Java website (http://java.sun.com or http://www.oracle.com/technetwork/java/index.html).

`Arrays.toString(<an array>)`

takes an array of values of a primitive or non-primitive data type as its input. It returns a String representation corresponding to the input array.

`Arrays.equals(<an array>,<another array of the same data type>)`

takes two arrays with the same data type as its input. It returns **`true`** if the two arrays are equal to one another. Otherwise, it returns **`false`**.

`Arrays.fill(<an array>,<a value of the same data type>)`

takes an array as its first input argument and a value whose data type must be the same as the elements in the array as the second. It does not return any values but assigns the value of the second input argument to every element in the array referred to by the first input argument.

```
Arrays.fill(<an array>,
            <first index>,<last index>,
            <a value of the same data type>)
```

works similarly to the previous *fill()* method but the value is assigned to the array only in the range starting from the position

specified by an **int** value **<first index>** to the position specified by another **int** value **<last index>**.

Arrays and Methods

Passing an array to a method

Just like variables of other data types, array variables can be used as a parameter in methods' argument lists. One thing, you need to keep in mind is that the actual value that is kept in an array variable is the reference to its associated array stored somewhere in the memory. Therefore, when an array variable is used as an input parameter to a method, that reference is copied to the corresponding array variable defined in the head of that method's definition.

Example 74: Passing arrays to methods

Consider the following program.

```
public class ArraysAndMethods
{
        public static void main(String[] args)
        {
            int [] a = {100,101,102,103};
            int k = 100;
            printArrayValues(a);
            System.out.println("k = "+k);
            someMethod(k,a);
            printArrayValues(a);
            System.out.println("k = "+k);
        }
        public static void someMethod(int k,int [] b){
            System.out.println("-------------In the method.");
            k = 0;
            for(int i=0;i<b.length;i++) b[i]=0;
            printArrayValues(b);
            System.out.println("k = "+k);
            System.out.println("---Going out of the method.");
        }
        public static void printArrayValues(int [] a){
            for(int i=0;i<a.length;i++)
                System.out.print(a[i]+",");
```

(continued on next page)

(continued from previous page)

```
            System.out.println();
        }
}
```

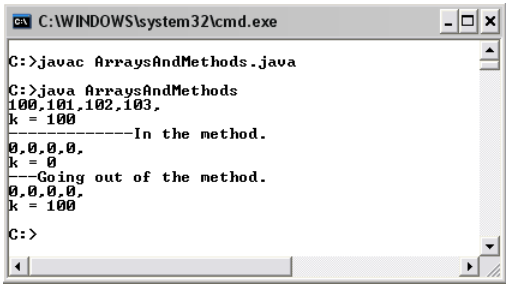


Figure 151: Passing arrays as input to methods

From the above example, notice the values of the array variable **a** and the **int** variable **k** before and after *someMethod()* is called. As we have discussed in the last chapter, the value of **k** in *main()* does not change when the value of the variable **k** in *someMethod()* is changed. (If this is not clear to you, go back and consult Chapter 8.) However, the value of the integers in the array associated with **a** are changed since the array variable **b** in *someMethod()* refers to exactly the same array as **a**. Therefore, **b[i]**, for each **i** equals 0 to 3, is the same value as **a[i]** in *main()*.

The method *printArrayValues()* serves as another example of how to define a method that takes an array as its input. Note that this method only aims at listing the elements of the input array. It could be replaced with the usage of *Arrays.toString()* without any problems.

Returning an array

An array can also be used as a returned value from a method. To do so, the return type must be specified in the method header appropriately as it is done with the values of other data types. To indicate that an array of a specific data type will be returned from a method, the return type must be the data type of the array element followed by square brackets.

For example,

```
public static int [] f(){
    // Body of f()
}
```

indicates that *f()* returns an array of **int**.

Example 75: Generating an array with random elements

The program listed below shows an example of a method that returns an array of **int**. The method generates an array with the length specified by an input argument, **length**. Each element in the array is an integer randomly chosen from **min** to **max** specified by the input argument passed to the method.

```
import java.util.Arrays;                                              1
public class GenRandomArray {                                         2
    public static void main(String [] args){                          3
        int len = 10;                                                 4
        int min = 0;                                                  5
        int max = 99;                                                 6
        int [] randoms = genRandomIntArray(len,min,max);              7
        System.out.println(Arrays.toString(randoms));                 8
    }                                                                 9
    public static int [] genRandomIntArray(                          10
                          int length,int min, int max){               11
        int [] a = new int[length];                                   12
        for(int i=0;i<length;i++){                                    13
            a[i] = (int)Math.round(Math.random()*(max-min)+min);      14
        }                                                             15
        return a;                                                     16
    }                                                                 17
}                                                                     18
```

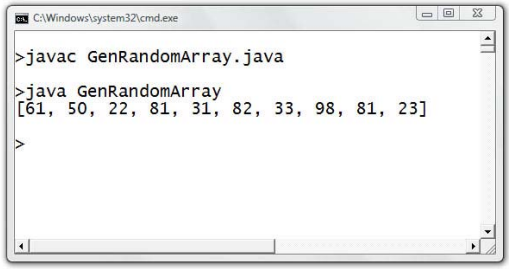


Figure 152: Generating an array with random integers

Example 76: Finding the maximum and the minimum array elements

The following Java program shows an example of methods that find the maximum and minimum values of an array.

```
public class MinMaxDemo                                                    1
{                                                                          2
  public static void main(String[] args)                                  3
  { int [] a = {-128,65,-235,99,0,26};                                     4
    int minIdx = findMinIdx(a);                                            5
    int maxIdx = findMaxIdx(a);                                            6
    System.out.println("min value is a["+minIdx+"]="+a[minIdx]);           7
    System.out.println("max value is a["+maxIdx+"]="+a[maxIdx]);           8
  }                                                                        9
  public static int findMinIdx(int [] a){                                  10
    int k, minIdx=0;                                                       11
    for(k=1;k<a.length;k++){                                               12
      if(a[k]<a[minIdx])                                                   13
        minIdx = k;                                                        14
    }                                                                      15
    return minIdx;                                                         16
  }                                                                        17
  public static int findMaxIdx(int [] a){                                  18
    int k, maxIdx=0;                                                       19
    for(k=1;k<a.length;k++){                                               20
      if(a[k]>a[maxIdx])                                                   21
        maxIdx = k;                                                        22
    }                                                                      23
    return maxIdx;                                                         24
  }                                                                        25
}                                                                          26
```

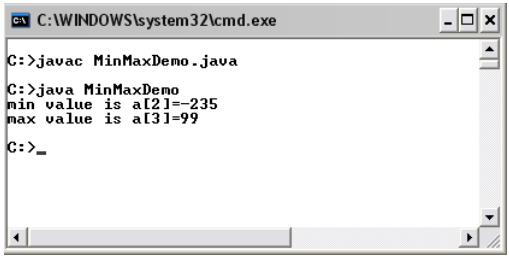


Figure 153: Finding the minimum and the maximum values in an array

findMinIdx() and *findMaxIdx()* defined on line 10 and line 18 are used for finding the index of the minimum and the maximum values in an array

input to the methods. The minimum value is located by first assuming that the element at the 0^{th} index is the minimum value, then iterating through each element and compare each value with the minimum value. If any element is smaller than the current minimum value, replace the minimum value with that element. This way, when the comparison has been performed with every element in an array, the minimum value obtained in the end is the actual minimum value in that array. The maximum value is located using a rather similar procedure.

Both the minimum and maximum values can be found using a single method, if we let the method return an array containing the indexes of the two values. Each index is placed in each position of the returned array. This is shown in the program listed below.

```
public class MinMaxDemo2
{
  public static void main(String[] args)
  { int [] a = {-128,65,-235,99,0,26};
    int [] idx = new int[2];
    idx = findMinMaxIdx(a);
    System.out.println("min value is a["+idx[0]+"]="+a[idx[0]]);
    System.out.println("max value is a["+idx[1]+"]="+a[idx[1]]);
  }
  public static int [] findMinMaxIdx(int [] a){
    int k, minIdx=0, maxIdx=0;
    for(k=1;k<a.length;k++){
                if(a[k]<a[minIdx])minIdx = k;
                if(a[k]>a[maxIdx])maxIdx = k;
    }
    int [] idx = {minIdx,maxIdx};
    return idx;
  }
}
```

In this program, findMinMax() returns an array of two integers in which the first position contains the index of the minimum value and the second position contains the index of the maximum value. The elements in the returned array can then be used accordingly.

'String [ ] args' Demystified

Now that we knows Java syntaxes related to arrays as well as how a method is defined, it is a good time to look at the expression **`String [] args`** appearing in the header of *main()*. We know from the last chapter that this expression is the declaration of a local variable to be used inside the method and the initial value of this variable will be passed along from the caller of this method. The value copied to this variable **`args`** is an array of *String* obtained from the command prompt when *java.exe* is called in order to execute the program. Elements of this array of *String* are called *command-line arguments*. It is a convenient way to receive the user's input without going through implementing custom prompts in the programs. When a Java program is executed, command-line arguments will be typed after the program name before the 'Enter' key is pressed. Multiple command-line arguments are separated with whitespaces before being arranged into an array of String passing along to the *main()* method, in which the program's instructions are contained.

Example 77: Command line summation utility

The following is a simple Java program that takes any number of command-line arguments and sums them together. We assume that every argument can be converted to a **`double`** value without any problems.

```
public class CommandLineSum {                                    1
    public static void main(String [] args){                    2
        double sum = 0.0;                                        3
        for(int i=0;i<args.length;i++){                          4
            sum += Double.parseDouble(args[i]);                  5
        }                                                        6
        System.out.println(sum);                                 7
    }                                                            8
}                                                                9
```

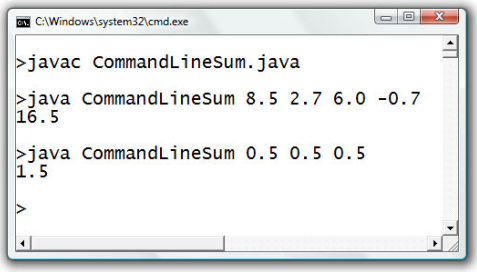


Figure 154: A program summing numbers read from the command line

Notice that we use **`args.length`** to obtain the number of command-line argument and **`args[i]`** to access the element at the **`i`**th index of the array just similar to other arrays.

Sequential Search

We usually need an ability to search for data of a particular value in an array. Sequential search is a search algorithm that operates by checking every element of an array one at a time in sequence until a match is found. A flow diagram of sequential search can be shown in Figure 155.

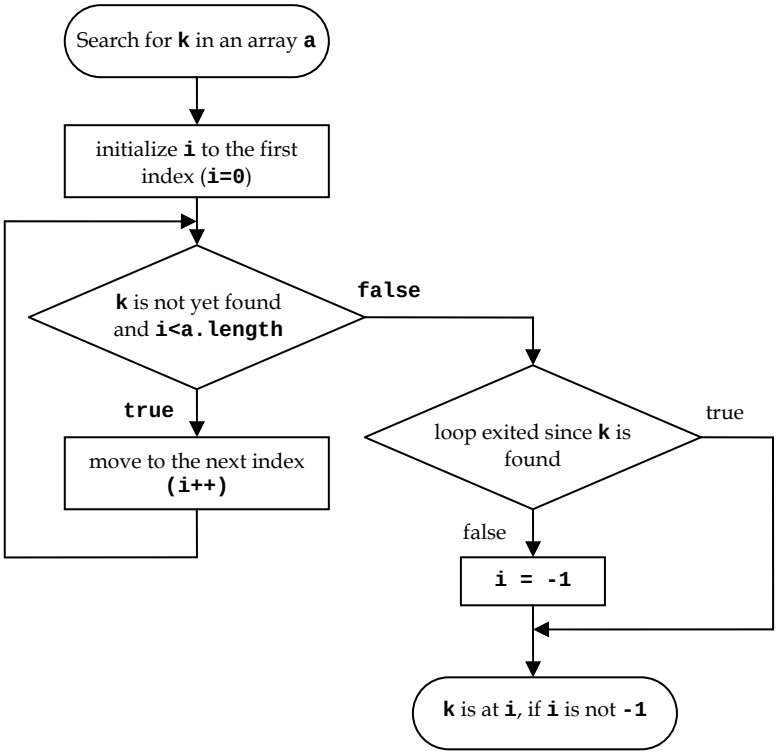


Figure 155: The sequential search algorithm

Suppose we want to search for a value `k` in an array `a`, we start checking at `a[i]` when `i=0`. If `a[i]` is not `k`, we increase `i` by one and repeat the checking. The process is iterated until `a[i] = k`, or until there is no more element in `a` (i.e. `i` exceeds `a.length-1`). Once the loop is exited, we have to check whether it is exited because k is found or there are no more elements to search. If the former case happens, `i` will still be less than `a.length`. If the latter case happens, `i` will be greater than or equal to `a.length`. In this case, we assign -1 to `i` to indicate that `k` is not found in `a`.

A Java method for performing the sequential search for an `int` value `k` in an array `a` could look like the one below.

```
public static int seqSearch(int [] a, int k){
        int i = 0;
        int len = a.length;
        while(i<len && a[i]!=k){
                i++;
        }
        if(i>=len) i=-1;
        return i;
}
```

Example 78: Sequential search

The following program demonstrates how the sequential search method can be used.

```
import java.util.Arrays;                                                    1
public class SeqSearchDemo                                                  2
{                                                                           3
  public static void main(String[] args)                                   4
  { int [] a = {99,105,86,34,108,25,11,96};                                 5
    System.out.println("a="+Arrays.toString(a));                           6
    System.out.println("86 is found at a["+seqSearch(a,86)+"]");           7
    System.out.println("96 is found at a["+seqSearch(a,96)+"]");           8
    System.out.println("0 is found at a["+seqSearch(a,0)+"]");             9
  }                                                                         10
  public static int seqSearch(int [] a, int k){                            11
    int i = 0;                                                              12
    int len = a.length;                                                     13
    while(i<len && a[i]!=k){                                                14
      i++;                                                                  15
    }                                                                       16
    if(i>=len) i=-1;                                                        17
    return i;                                                               18
  }                                                                         19
}                                                                           20
```

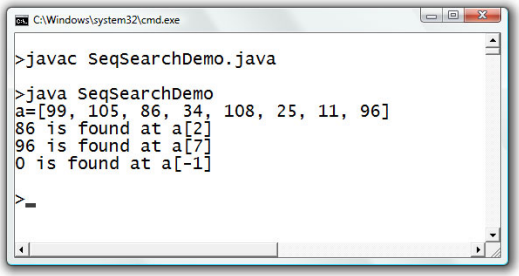


Figure 156: Demonstration of the sequential search

In the program, *seqSearch()* is called on line 7, line 8, and line 9. The sequential search algorithm is implemented in *seqSearch()* whose definition is listed on line 11 to line 19.

Selection Sort

Sometimes we need to sort the values appeared in an array. Algorithms performing the task are called sorting algorithms. Probably the most intuitive sorting algorithm is selection sort, in which the array is traversed several times. Each time of the traversal, the maximum or the minimum element will be identified and placed in its correct position. Then, the elements that have already been placed in their correct positions will be ignored in later traversals.

To sort the values in an array increasingly, selection sort works as follows:

1. Let `k` be the first position of the array (i.e. `k = 0`).
2. Find the minimum value of the portion of the array from the k^{th} position to the last position.
3. Swap the minimum value with the value in the k^{th} position.
4. Increase `k` by one.
5. Repeat step 2 to step 4 until `k` reaches the end of the array.

Important points that we can gather from the steps above are:

- If the number of elements in the array is *len*. The total number of traversal made is *len* – 1.
- For the i^{th} traversal (where i runs from 1 to n – 1), the portion of the array to be traversed is from the $(i\text{-}1)^{th}$ (which is the k^{th})

position to the (len-1)th position. Therefore, the minimum value found in the i^{th} traversal is the minimum elements inside that corresponding portion of the array.

- The value of `k` is always the position in which the original element prior to the latest traversal will be swapped with the minimum value found in that traversal.

- After the i^{th} traversal, the first i slots of the array (from the 0th to (i-1)th position) will be sorted.

- After the (len-1)th (last) traversal, the first len-1 slots of the array is sorted which effectively means that the last single unsorted element is also in its correct position. Therefore, the sorting can stop here.

Let's look at an illustrated example of the sorting of an array `a = {1, -2, 4, 3}` increasingly in Figure 157.

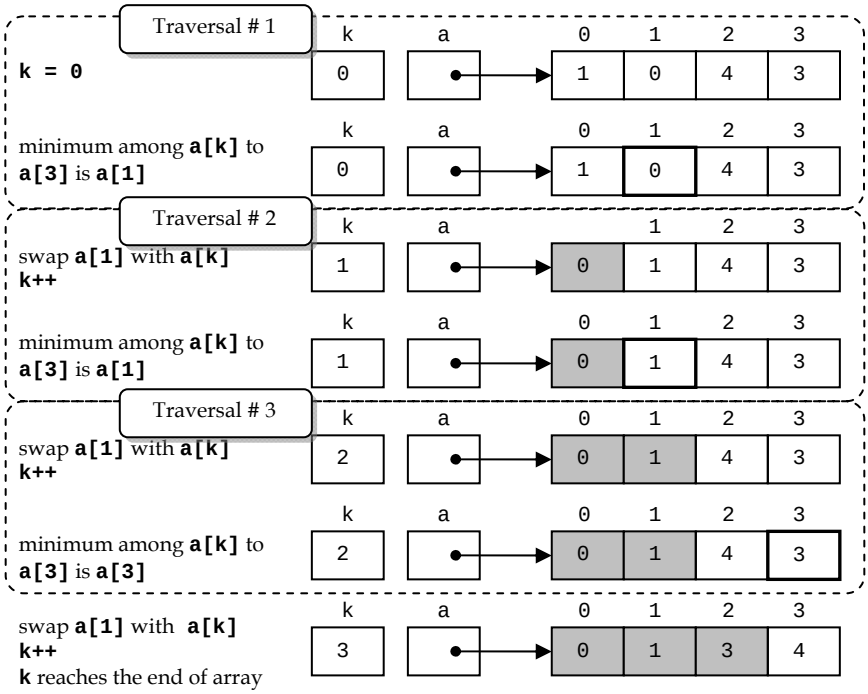


Figure 157: An example of performing the selection sort on an array. Each traversal only traverses the portion of the array that is not shaded.

If the sorting is to be done decreasingly, the maximum element associated with each traversal should be located and swapped with the element in the **k**th position instead of the minimum element.

Another alternative modification to make the algorithm sorts the elements decreasingly could be that we still look for the minimum element but instead of running the value of **k** from the front of the array to the back, **k** should be run from back to front.

The following method is a possible implementation of the selection sort algorithm on an array of **int**. The method also takes the second input argument as an argument to select whether the sorting should be performed increasingly or decreasingly. Note that the method depends

on other methods that are responsible for finding the position of the minimum element or the position of the maximum element.

```
public static void selectionSort(double [] d,boolean inc){    1
    int idxToBeSwapped, len = d.length;                       2
    double temp;                                              3
       for(int i=1;i<=(len-1);i++){                           4
        int k = i-1;                                          5
        if(inc){                                              6
            idxToBeSwapped = findMinIdx(d,k);                 7
        }else{                                                8
            idxToBeSwapped = findMaxIdx(d,k);                 9
        }                                                     10
        temp = d[idxToBeSwapped];                             11
        d[idxToBeSwapped] = d[k];                             12
        d[k] = temp;                                          13
       }                                                      14
}                                                             15
```

The *selectionSort()* method is implemented so that it looks for the minimum element in the portion of the input array according to a given traversal and place it at the front of the portion when it sorts the array increasingly. While it sorts the array decreasingly, the maximum element is looked for instead of the minimum element. The statement on line 5 of *selectionSort()* moves **k** forward as each iteration of the for loop has passed. **findMinIdx(d,k)** and **findMaxIdx(d,k)** are the methods that find the position of the maximum and the minimum elements in the portion of the array starting from **k** to the last element respectively. Implementations of these two methods are shown below. The statements on line 11 to line 13 are there to swap the values between the element at the $\mathbf{k}^{th}$ position with the position returned by either *findMinIdx()* or *findMaxIdx()*.

```
public static int findMinIdx(double [] d,int k){              1
    int minIdx = k;                                           2
    for(int i=k+1;i<d.length;i++){                            3
        if(d[i] < d[minIdx]){                                 4
            minIdx = i;                                       5
        }                                                     6
    }                                                         7
    return minIdx;                                            8
}                                                             9
```

```
    public static int findMaxIdx(double [] d,int k){
        int maxIdx = k;
        for(int i=k+1;i<d.length;i++){
            if(d[i] > d[maxIdx]){
                maxIdx = i;
            }
        }
        return maxIdx;
    }
```

Example 79: Selection sort

Below is a program that uses the selection sort algorithm to sort the numbers supplied via the command line with a menu to select the direction of the sorting. Note that the method definitions of *selectionSort(), findMinIdx()* and *findMaxIdx()* are similar to what are presented earlier and are omitted from the code listing in this example.

```
import java.io.*;
import java.util.Arrays;
public class SelectionSortConsoleUI {
    public static void main(String [] args) throws IOException{
        BufferedReader in =
                new BufferedReader(
                    new InputStreamReader(System.in)
                );
        boolean inputOK = false, inc=true;
        String input;
        System.out.println("\nInput ->"+Arrays.toString(args));
        while(!inputOK){
            System.out.print("Sort increasingly");
            System.out.print(" or decreasingly (d/i):");
            input = in.readLine();
            if(input.length()==1){
                switch(input.charAt(0)){
                    case 'd':
                        inc = false;
                        inputOK = true;
                        break;
                    case 'i':
                        inc = true;
                        inputOK = true;
                        break;
                    default:
                        inputOK = false;
                }
            }
```

(continued on next page)

(continued from previous page)

```
        }                                                          30
        double [] d = new double[args.length];                     31
        for(int i=0;i<args.length;i++){                            32
            d[i] = Double.parseDouble(args[i]);                    33
        }                                                          34
        selectionSort(d,inc);                                      35
        System.out.println("Sorted->"+Arrays.toString(d)+"\n");    36
    }                                                              37
    public static void selectionSort(double [] d,boolean inc){…}
    public static int findMinIdx(double [] d,int k){…}
    public static int findMaxIdx(double [] d,int k){…}
}
```

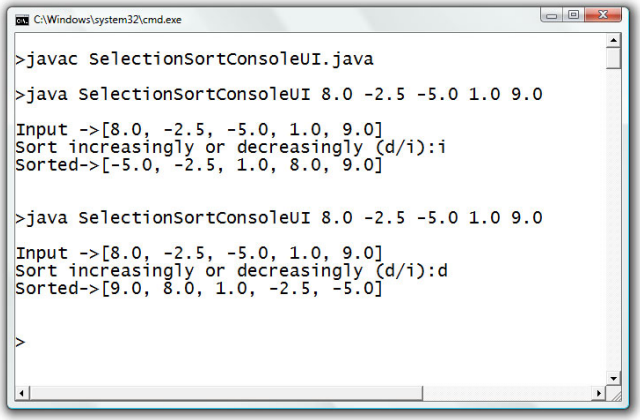


Figure 158: A program utilizing the selection sort to sort its inputs

This program reads the data to be sorted from the command line and prompts for whether the user would like to sort the data increasingly or decreasingly. After checking the user's choice, it then converts each element in **`args`** into a double value and put it in an array of double which is consequently sorted by the *selectionSort()* method.

Multi-dimensional Arrays

An element contained in an array can be another array itself. An array in which each element is another array is called *multi-dimensional array*. Figure 159 illustrates an array of arrays of integers, or a two-dimensional array of integers.

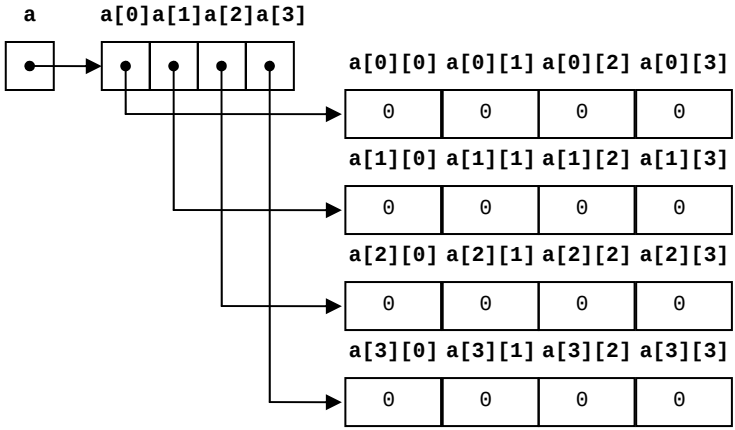


Figure 159: An array of arrays of int

A multi-dimensional array of a certain data type is declared by inserting pairs of [] after the data type specification. The number of [] equals the dimension of the array. Consider the following statements.

```
String [][] arrayOfString;
double [][][] a3DArray;
```

The first statement declares a variable named **`arrayOfString`** as a two-dimensional array of *String*. The other statement declares another variable named **`a3DArray`** as a three-dimensional array of **`double`**.

The following statements show how to declare and initialize multi-dimensional arrays with default values according to their data types.

```
int [][] k = new int[3][5];
```

```
boolean [][][] p = new boolean[2][2][2];
```

The first statement declares a variable **k** and assigns to it a reference to an array of length 3, each of whose elements is a reference to an array of five integers. All elements in the arrays of integers are initialized to 0.

The second statement declares a variable **p** and assigns to it a reference to an array of length 2, each of whose elements is a reference to a two-dimensional array of **boolean** value of the size 2 x 2. All **boolean** values are initialized to **false**.

Figure 160 shows an illustration of the variable **p**.

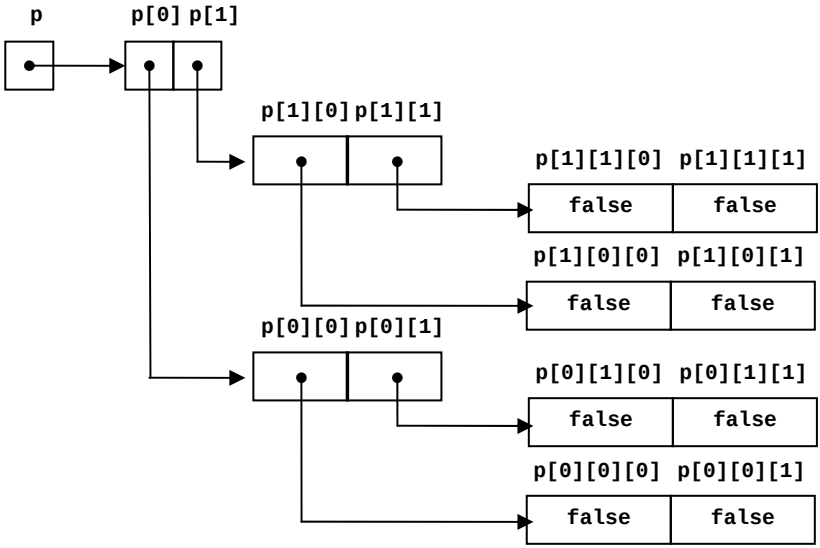


Figure 160: A three dimensional array of boolean

Initializer Lists for Multi-dimensional Arrays

Nested initializer lists can be used to initialize a multi-dimensional array with values other than the default value. For example, the statement:

```
int [][] k = {{1,2},{3,4,5},{8,10,12,14}};
```

initializes the variable **k** to refer to an array of three elements, where the first element is an array of two **int** values **{1,2}**, the second element is an array of three **int** values **{3,4,5}**, and the last element is an array of four **int** values **{8,10,12,24}**.

To access elements in **k**, we use these indexing mechanisms:

k : refers to the whole two-dimensional array of **int**.
k[0] : refers to the array **{1,2}**.
k[1] : refers to the array **{3,4,5}**.
k[2] : refers to the array **{8,10,12,14}**.
k[i][j] : refers to the **j**th element of **k[i]**.

E.g. **k[0][1]** equals 2, **k[1][0]** equals 3, and **k[2][3]** equals 14 etc.

The following statement shows an example of using an initializer list to initialize a three-dimensional array of *String*.

```
String [][][] s =
    {
        {{"Hamburg","Berlin","Munich"},{"Paris","Dijon"}},
        {{"Hanoi"},{"Bangkok","Chiang Mai"}}
    };
```

Being initialized this way,

s : refers to the whole three-dimensional array of *String*.
s[0] : refers to the two-dimensional array **{{"Hamburg", "Berlin", "Munich"}, {"Paris", "Dijon"}}**.
s[1] : refers the two-dimensional array **{{"Hanoi"},{"Bangkok","Chiang Mai"}}**
s[0][0] : refers to the array **{"Hamburg", "Berlin", "Munich"}**.
s[0][1] : refers to the array **{"Paris", "Dijon"}**.
s[1][0] : refers to the array **{"Hanoi"}**.
s[1][1] : refers to the array **{"Bangkok","Chiang Mai"}**.
s[i][j][k] : refers to the **k**th element of **s[i][j]**.

E.g. **s[0][1][1]** equals **"Dijon"**, **s[1][1][0]** equals **"Bangkok"**, and etc.

Example 80: Lengths of multi-dimensional arrays

Let's look at an example program demonstrating the indexing of multi-dimensional array. Nested **for** loops are used for browsing through each level of the three-dimensional array of **int** named **a**. Pay attention to the length of **a[i]**, **a[i][j]** and the value of each **int** value **a[i][j][k]**.

```
public class ArrayLengthDemo                                          1
{                                                                      2
  public static void main(String[] args)                              3
  {                                                                    4
      int [][][] a = {{{1,2,3},{4,5},{6}},{{7,8},{9}}};               5
      System.out.println("a.length = "+a.length);                      6
                                                                       7
      for(int i=0;i<a.length;i++){                                     8
          System.out.println("a["+i+"].length = "+a[i].length);        9
                                                                       10
          for(int j=0;j<a[i].length;j++){                              11
              System.out.print("a["+i+"]["+j+"].length = ");           12
              System.out.println(a[i][j].length);                      13
                                                                       14
              for(int k=0;k<a[i][j].length;k++){                       15
                  System.out.print("a["+i+"]["+j+"]["+k+"]=");        16
                  System.out.print(a[i][j][k]+", ");                   17
              }                                                        18
              System.out.println();                                    19
          }                                                            20
      }                                                                21
  }                                                                    22
}                                                                      23
```

The output of the program should prints the following text.

```
a.length = 2
a[0].length = 3
a[0][0].length = 3
a[0][0][0]=1, a[0][0][1]=2, a[0][0][2]=3,
a[0][1].length = 2
a[0][1][0]=4, a[0][1][1]=5,
a[0][2].length = 1
a[0][2][0]=6,
a[1].length = 2
a[1][0].length = 2
a[1][0][0]=7, a[1][0][1]=8,
a[1][1].length = 1
a[1][1][0]=9,
```

Readers should try to draw a picture of the array referred to be the variable **a** from the initializer list on line 5 and see whether it is consistent with what are printed out by the program in this example.

Let's finish this chapter with a program that solves a matrix-related calculation which Science and Engineering students will surely come across in college.

Example 81: The n^{th} power of a matrix

We would like to write a program that can calculate the result of $\mathbf{A}^n$ where **A** is a square matrix and n is a positive integer.

Problem definition: The program needs to calculate the n^{th} power of a matrix whose elements, as well as the value of n, are specified by the user.

Analysis: Elements of **A** and the power n should be read from keyboard. The result of the calculation should be shown as the output on screen.

Design:

- The user must specify the size of the square matrix **A** via keyboard. The dimension will be kept in an `int` variable named `dim`.
- Elements of **A** will be kept in a two-dimensional array named `a`. With the size of **A** known, the program should iteratively prompt the user to input elements of **A** one by one. Each element will be stored in `a`.
- The program prompts the user to enter n via keyboard. The input will be kept in `n`.
- $\mathbf{A}^k$ can be calculated from $\mathbf{A}^{k-1} \times \mathbf{A}$ for k = `2, 3, 4 , … , n`. That means $\mathbf{A}^n$ can be calculated by iteratively multiply **A** with the result of the multiplication prior to the current iteration. After each iteration of the multiplication, use another two-dimensional

array named **b** to store $\mathbf{A}^{k-1}$. Also, use another two-dimensional array named **c** to store the result.

- To calculate $\mathbf{C} = \mathbf{B} \times \mathbf{A}$. Use the relation: $c_{ij} = \sum_{k=1}^{n} b_{ik} a_{kj}$
- Show each element of $\mathbf{A}^n$ on screen.

Implementation:

```
import java.io.*;                                                    1
public class MatrixPower                                             2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {   // Declare variables                                          5
      double [][] a, b, c;                                           6
      int dim, n;                                                    7
      BufferedReader stdin =                                         8
          new BufferedReader(                                        9
              new InputStreamReader(System.in));                     10
      // Read matrix size                                            11
      System.out.print("Enter matrix size:");                       12
      dim = Integer.parseInt(stdin.readLine());                      13
      // Create a, b, c and read each element of a                   14
      a = new double[dim][dim];                                      15
      b = new double[dim][dim];                                      16
      c = new double[dim][dim];                                      17
      for(int i=0;i<dim;i++){                                        18
          for(int j=0;j<dim;j++){                                    19
              System.out.print("a"+(i+1)+(j+1)+"=");                20
              a[i][j] = Double.parseDouble(stdin.readLine());        21
          }                                                          22
      }                                                              23
      // Read power n                                                24
      System.out.print("Enter n:");                                  25
      n = Integer.parseInt(stdin.readLine());                        26
      // Perform raising a to the n th power                         27
      b = a;                                                         28
      for(int k=1;k<n;k++){                                          29
          c = multSqMatrices(b,a);                                   30
          b = c;                                                     31
      }                                                              32
      // Show the result on screen                                   33
      showMatrix(c);                                                 34
  }                                                                  35
                                                                     36
  public static double [][] multSqMatrices(                          37
                              double [][] b,double [][] a){          38
```

(continued on next page)

(continued from previous page)

```
      int dim = b.length;                                          39
      double [][] c = new double[dim][dim];                        40
      for(int i=0;i<dim;i++)                                       41
          for(int j=0;j<dim;j++)                                   42
              for(int k=0;k<dim;k++)                               43
                  c[i][j] = c[i][j]+b[i][k]*a[k][j];               44
      return c;                                                    45
  }                                                                46
                                                                   47
  public static void showMatrix(double [][] c){                    48
      int nRows = c.length;                                        49
      int nCols = c[0].length;                                     50
      for(int i=0;i<nRows;i++){                                    51
          for(int j=0;j<nCols;j++){                                52
              System.out.print(c[i][j]+"\t");                      53
          }                                                        54
          System.out.println();                                    55
      }                                                            56
  }                                                                57
}                                                                  58
```

Program testing: Figure 161 shows some screenshots when the program is executed with some data.

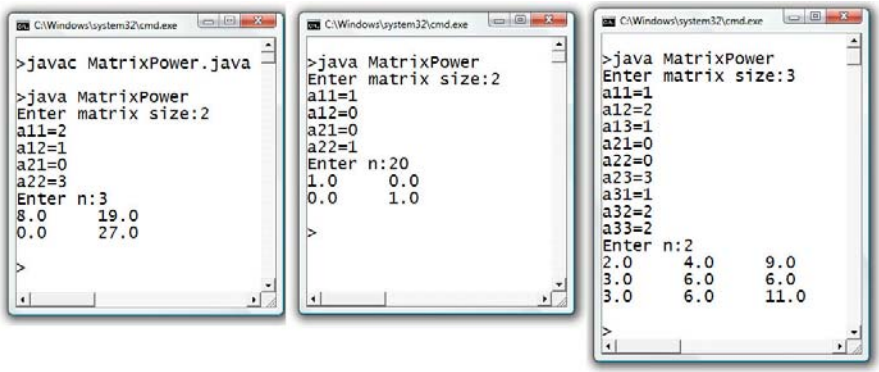


Figure 161: Some outputs of a program finding powers of matrices

Let's try computing the square of the matrix input to the program in the left of Figure 161.

$$\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}=\begin{bmatrix}2\times2+1\times0 & 2\times1+3\times1\\0\times2+3\times0 & 0\times1+3\times3\end{bmatrix}\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}$$
$$=\begin{bmatrix}4 & 5\\0 & 9\end{bmatrix}\begin{bmatrix}2 & 1\\0 & 3\end{bmatrix}=\begin{bmatrix}4\times2+5\times0 & 4\times1+5\times3\\0\times2+9\times0 & 0\times1+9\times3\end{bmatrix}$$
$$=\begin{bmatrix}8 & 19\\0 & 27\end{bmatrix}$$

Also, the input in the second screenshots shows the case when the identity matrix is raised to the power of 20 which also results in the identity matrix. It is left to the readers to verify the third case.

Anyway, the program seems to work well. Please be reminded that precisely testing the program for its logical correctness is beyond the scope of this book.

Exercise

1. Show how to declare variables corresponding to the following:
    a. An array of **`int`**.
    b. An array of **`boolean`**.
    c. An array of **`String`**.
    d. An array of arrays of **`double`**.
    e. A two-dimensional array of *Rectangle*.
    f. A three-dimensional array of **`char`**.
2. Declare and initialize arrays corresponding to the following:
    a. An array of 20 **`int`** values.
    b. An array of double where its length equals the length of an array of **`int`** called **`b`**.
    c. An array of **`boolean`** where its first three values are true and the other two are false.
    d. An array of *String* containing the names of the seven days in a week.
    e. An array containing an array of 1.0, 2.5, 3.0, and another array of 2.5, 3.0, 4.5.

f. A two-dimensional array suitable for representing an identity matrix of the size 3×3

3. Is it valid to create an array where each element is an array whose length is different from the lengths of other elements in the same array.

4. Determine the output of the following code segment.

```
int [] a = new int[10];
a[1] = 2;
a[a.length-1]=8;
for(int i=0;i<a.length;i++){
        System.out.print(a[i]+"\t");
}
```

5. Determine the output of the following code segment.

```
int [] a = new int[10];
for(int i=0;i<a.length-1;i++){
        a[i] = a[++i]+i;
}
for(int i=0;i<a.length;i++){
        System.out.print(a[i]+"\t");
}
```

6. Determine the output after *main()* is executed.

```
public static void main(String[] args) {
        int k  = 1;
        int [] a = {10,11,12,13,14};
        f(k,a);
        System.out.println(k);
        showArrayContent(a);
}
public static void f(int k,int [] b){
        if (k >= b.length) return;
        for(int i=k;i<b.length;i++){
            b[i]=b[b.length-i];
        }
        k = 0;
}
public static void showArrayContent(int [] a){
        for(int i=0;i<a.length;i++)
            System.out.println(a[i]);
}
```

7. Write a method that receives an array of **int** and returns the sum of every element in the array.

8. Write a method that evaluates the value of a polynomial function *p(x)* at given values of *x*. The function is of the form $c_n x^n + c_{n-1}x^{n-1} + \ldots + c_0$. The method header is given as:

```
public static double [] p(double [] x, double [] coeff)
```

   The k^{th} element in **coeff** is corresponding to the c_k. Each element in the returned array of **double** is the value of *p(x)* evaluated at *x* being the value of the element in **x** at the same index.

9. Write a method called *findRep()* whose header follows:

```
public static int findRep(int [] a, int target, int nRep)
```

   The method finds whether **a** contains **nRep** consecutive elements whose values equal **target** or not. If so, it returns the position of the first element whose value equals **target**. Otherwise, it returns -1. For example, if **a** contains 6, 8, 9, 9, 9, 3, 9, 2, 0, **findRep(a,9,2)** and **findRep(a,9,3)** return 2, while **findRep(a,9,4)** and **findRep(a,10,1)** return -1.

10. Write a method that sorts an input array of **int** in place (i.e. the elements in the original array are sorted. There is no new array resulted from the sorting). There must be another parameter determining whether to sort this array increasingly or decreasingly.

11. Every method in this problem receives two arrays of **int** as their input parameters.

    a. Write *combine()* which returns a new array whose elements are taken from both input arrays and their orders are preserved starting from the elements from the first input array followed by the ones from the second.
    b. Write *union()* which returns a new array whose elements are unique elements taken from both input

arrays. The elements of the output array should be sorted increasingly.

c. Write *intersect()* which returns a new array where every elements in the array must be unique and appear in both input arrays. The elements of the output array should be sorted increasingly.
d. Write *subtract()* which returns a new array whose elements are unique and appear in the first input array but not in the second one. The elements of the output array should be sorted increasingly.
e. Write *xor()* which returns a new array whose elements are unique and appear in either one of the input arrays but not both. The elements of the output array should be sorted increasingly.

12. Suppose that two arrays of **`int`** are said to be equal if they have similar lengths and every elements in the same positions of the two arrays are equal. Write a method called *isEqual()* which returns **`true`** if its two input arrays equal and **`false`** otherwise.

13. Repeat the previous problem in the case where positions do not matter, i.e. the two arrays are said to be equal if their elements form sets with similar members.

14. Write a method that receives an array of *String* , together with a *String* and returns **`true`** if there is at least one element of the input array that contains or equals the other *String* input. Otherwise, it returns **`false`**.

15. Explain why the following code segment lead to a failed compilation.

```
final double RANGE = 200;
int step = 12, k=6;
int [][] a = new int[(int)(RANGE/step)][k];
for(int i=0;i<a.length;i++)
        a[i] = new int[2][2];
```

16. What is the output of the following code segment?

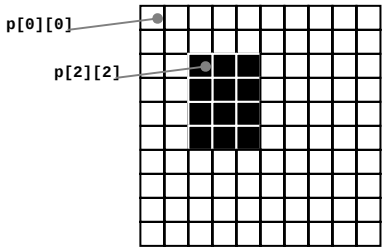
```
String [][][] x = new String[5][6][7];
System.out.println(x.length*x[2].length);
```

17. One way to represent a black-and-white image is to store **boolean** values in a two-dimensional array, **p**. **p[i][j]** is **true** if the pixel in the **i**th row and **j**th column is black. Similarly, it is **false** if the corresponding pixel is white.

    Write a method:

    ```
    public static boolean drawRect(boolean p, int x, int y,
    int width, int height, int ink)
    ```

    If ink equals 0, the method draws a white rectangle whose topleft corner locating at **p[x][y]**. Its width and height are the values of **width** and **height** respectively. If **ink** equals 1, the method draws a black rectangle instead. If **ink** equals -1, the drawing is done in a way that every pixels of the rectangle drawn by the method are toggled from white to black, or black to white. The following array demonstrate an example of **p** after performing **drawRect(p,2,2,3,4,1)** on an all-white array **p** whose size is 10×10.

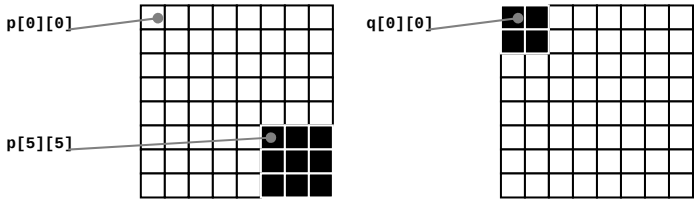


    The method returns **true** if the drawing is performed successfully. It returns false and does not perform any drawing when at least one of the following situations take place:

    - The value specified by ink is not -1, 0, ir 1.
    - The specified topleft corner does not fall in the vicinity of **p**, i.e. the value of **x**, **y** or both is not in the range of the array.
    - The rectangle exceeds the vicinity of **p**.

18. Repeat the previous problem. However, this time, the method should attempt to draw the specified rectangle even though the topleft corner does not fall in the vicinity of **p** or the whole rectangle does fit in **p**.

    For example, if **p** and **q** are of the size 8×8 and initially all zeros. **drawRect(p,5,5,2,5,1)** and **drawRect(q,-2,-2,4,4,1)** would result in the following arrays.



19. In a seat reservation program of an airline, the seating chart of an airplane is represented using a two-dimensional array, **seats**. The array **seats[i][j]** contains the name of the passenger who has reserved the seat number **j** in the **i+1**th row, where **j** is 0 for the seat number *A*, **j** is 1 for the seat number *B*, and so on. **seats[i][j]** stores **null** if the seat is vacant. Note that different airplanes may have different numbers of rows. However, assume that the number of seats in each row on the same airplane is constant.

    Write methods, that have one of their input arguments being the array **seats**, for performing the tasks in the following items. Decide on the names, their input arguments, and their returned values appropriately.

    a. Showing seating chart, labeled with row and seat numbers, by using '**.**' to represent a vacant seat and '**X**' to represent a reserved seat. An example of the seating chart could by like the one shown below.

```
    A B C D E F G H I L K L
1   X X X . . . . . . X X X
2   . X X . . . . . . . . .
3   . X X . . . . . . . . .
4   X . . . . . . X X X . .
5   . . . . . . . . . . . .
6   . . . . . . . . . . . .
7   . . . . X X X X . . . .
8   . . . . . . . . . . . .
9   . . . . . . . . . . . .
10  . . . . . . . . . . . .
11  . . . . . . . . . . . .
12  . . . . . . . . . . . .
```

If the caller of the method does not wish to see seating of every rows, the caller can indicate the range of the rows wished to be shown by specify the row numbers of the first row and the last row to be shown.

b. Adding a passenger name to a selected seat. It returns **`true`** if the operation is successful, and return **`false`** if the selected seat is not empty. Given that the selected seat is specified in the form of a *String* in the form: "[row number]-[seat number]", such as "1-A", "25-E", and "36-I". The method also checks whether the specified seat is in the valid range. Unless the specified seat is in the range of the array **`seats`**, the method does nothing and returns **`false`**.

c. Removing the passenger at a specified seat.

d. Searching for the seat reserved by a passenger by his/her name. The method returns the *String* representing the seat location or **`null`** if there is no passenger of the given name. Assume that each passenger can occupy only one seat at a time.

e. Counting the number of seats available in each row. The method returns an array of **`int`** where the value at the i^{th} index contains the number of available seats in the $i+1^{th}$ row.

f. Searching for available *n* consecutive seats in the same row. The method returns the *String* representing the left-most seat location of the available *n* consecutive seats in the front-most row that has such availability. If there is no such availability the method returns `null`.

g. Randomly relocating passengers in the seating chart. Each passenger must be assigned a seat not conflicting with other passengers. The method returns the randomized seating as its output, while the input seating stays intact. (This method is not going to be useful for any functioning airlines!)

Chapter 10: Recursive Problem Solving

Objectives

Readers should

- Be able to explain the concept of recursive definition
- Be able to use recursion in Java to solve problems

Recursive Problem Solving

To solve a problem using the recursive problem solving technique, we break that problem into identical but smaller, or simpler, problems and solve those smaller problems to obtain a solution to the original one. For example, we can find the summation of integers from 0 to a positive integer *n* by finding the summation of integers from 0 to a positive integer *n-1* first then add to that result the integer *n* to obtain the result of the original problem. Finding the summation of integers from 0 to a positive integer *n-1* is considered a similar to but smaller problem than finding the summation from 0 to *n*. Also, we know that if *n* equals 0, the result is just zero.

The following example implements the above idea in a Java method.

Example 82: Positive integer summation

Mathematically, we can write the summation of the first *n* positive integers as:

$$s(n) = s(n-1) + n$$

where $s(n)$ is the summation of integers from 0 to *n* for any positive integer *n*. Also, we know that $s(0) = 0$.

A Java method for finding such a summation could be written in a recursive fashion as in the following code segment.

```
public static int s(int n){                                    1
       if(n==0) return 0;                                      2
       return s(n-1)+n;                                        3
}                                                              4
```

In the body of *s()*, we can see that the method call itself but the input parameter used is smaller every time the method is called. If *n* equals 2, **`s(2)`** calls **`s(1)`**, wait for the value to be returned from the method, and adds *n*, which is now 2, to the returned value before returning the result to the caller. In a similar fashion, once **`s(1)`** is called, it invokes **`s(0)`**, wait for the value to be returned from the method, which is 0, and adds 1 to the returned value before returning the result to the **`s(2)`**.

The invocation explained can be depicted in the following picture.

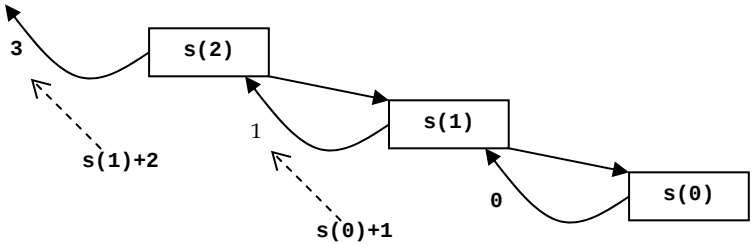


Figure 162: Finding s(n)=s(n-1)+n where n=2 recursively

Now let's apply the idea of the recursive problem solving to finding the factorial function.

Example 83: Factorial function

The factorial function of *n*, written as *n*!, which is the product of the first *n* positive integers can be described as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times \ldots \times 1 & \text{if } n > 0 \end{cases}$$

From the definition of the function, $(n - 1) \times (n - 2) \times \ldots \times 1$ is $(n - 1)!$. Therefore, the definition above can be rewritten as:

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1)! & \text{if } n > 0 \end{cases}$$

Again, $n!$ can be solved recursively from $(n\text{-}1)!$ and we know that 0! is 1. A Java method that can find $n!$ recursively can be written as the following code segment.

```
public static int factorial(int n){                    1
    if(n==0) return 1;                                 2
    return factorial(n-1)*n;                           3
}                                                      4
```

As we can see from line 3, given an input n, the method call itself but with an input one which is one smaller than the original. When the input reaches 0, the method just returns a value without further recursively calling itself.

The picture below depict the method invocation of **`factorial(4)`**.

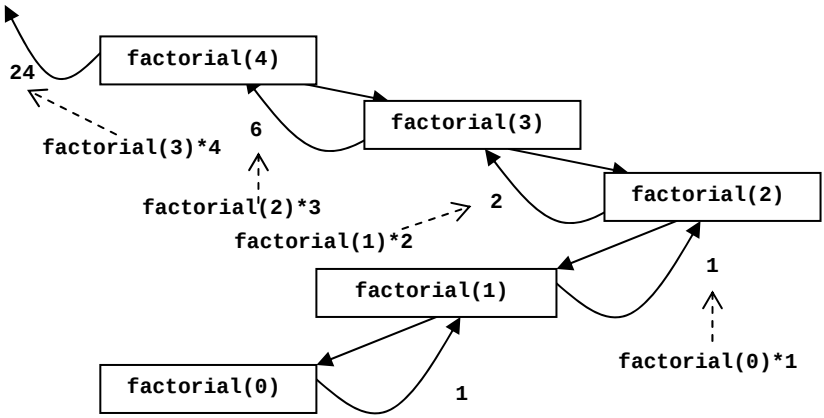


Figure 163: Finding 4! recursively

Solving the two problems shown in Example 82 and Example 83 can also be done using iterative constructs such as **`for`** loops. You should try writing the methods shown above using iterative approach and compare them with the recursive approach in various aspects, such as their lengths, their implementation complexity, as well as the difficulty in coming up with the solutions via both approaches.

Recursive Method Design

In earlier examples presented in this chapter, we use methods containing statements that call themselves in order to implement the recursive problem solving idea. These methods are called *recursive methods*.

A recursive method must have two parts. The first part determines the case where the recursive method invocation terminates. Cases where this part occurred are called the *base cases*. The other part recursively calls itself, but with simpler parameters. Cases where this part occurred are called the *recursive cases*. Each time the method is recursively called the parameters must be made simpler and must move towards the base cases. Failure to terminate recursive methods either from the missing of the base cases or the recursion never falls into the base cases results in an *infinite recursion* when the method is called during its associated program execution.

Example 84: Iterative to recursive

Implement the following method using the recursive approach.

```
public static int f(int n){                                    1
    int a = 0, b = 0, c = 0, i = 0;                            2
    while(i<n){                                                3
        c = b;                                                 4
        b = a;                                                 5
        a = 2*b+3*c+1;                                         6
        i++;                                                   7
    }                                                          8
    return a;                                                  9
}                                                              10
```

Let's first analyze the method for the base cases. Notice that the method returns the value of **a** which is initialized with 0 and it will never be changed if the program flow does not go into the **while** loop on line 3 to line 8. Such a condition is when the value of **i** is not less than **n**, which also means that the input **n** is greater than or equal to **i**. Therefore, the condition for the base cases which are cases when the program can return a value right away should at least include these cases and the value returned in the initial value of **a** which is 0.

Now, we turn to analyzing the **while** loop. Given the input **n** being a positive integer, statements inside the **while** loop will be performed **n** times. Figure 164 shows how the values of **a**, **b**, and **c** change as i increases. Notice that the value of **a** after the **k**th iteration is the result of the method when the method's input **n** is **k**. Also, the value of **b** in the **k**th iteration is the value of **a** in the **(k-1)**st iteration and the value of **c** in the **k**th iteration is the value of **a** in the **(k-2)**st iteration. Therefore, in the **k**th iteration, the statement **a = 2*b+3*c+1;** states that is the result of the method when the method's input **n** equals **k** is the summation three values which are:

- two times the result of the method when the method's input **n** equals **k-1**,
- three times the result of the method when the method's input **n** equals **k-2**, and
- the integer value of 1.

Consequently, the resulting iterative method could be written as:

```
public static int f(int n){
    if(n<=0) return 0;
    return 2*f(n-1)+3*f(n-2)+1;
}
```

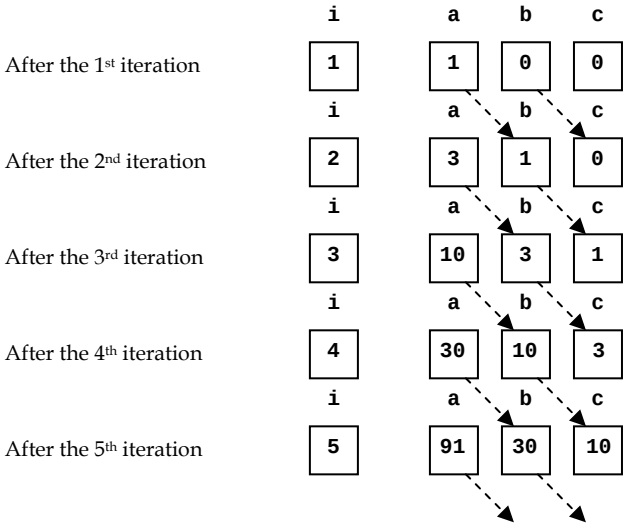


Figure 164: How variables change when the loop keeps iterating

---

Example 85: Fibonacci numbers

The *Fibonacci numbers* form a sequence of integer defined recursively by:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

where $F(n)$ is the n^{th} Fibonacci number for every non-negative integer n.

Therefore, the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, … . The Fibonacci numbers model many things in nature, such as branching in trees and arrangement of pine cones.

The following Java program prints out the first 20 Fibonacci numbers with the use of a recursive method *fibo()*.

```
public class FiboDemo                                                  1
{                                                                      2
        public static void main(String[] args)                        3
        {   final int n = 20;                                          4
            for(int i=0;i<20;i++)                                      5
                System.out.print(fibo(i)+",");                         6
            System.out.println();                                      7
        }                                                              8
        public static int fibo(int n){                                 9
            if(n<=0) return 0;                                         10
            if(n==1) return 1;                                         11
            return fibo(n-1)+fibo(n-2);                                12
        }                                                              13
}                                                                      14
```

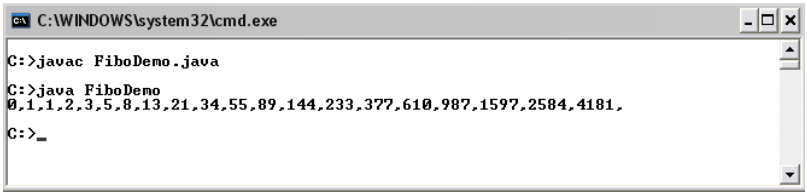


Figure 165: Finding Fibonacci numbers

The following picture depicts the invocation of *fibo()* in finding **`fibo(4)`**. The numbers listed in solid circles indicate the order of method invocations and value returning.

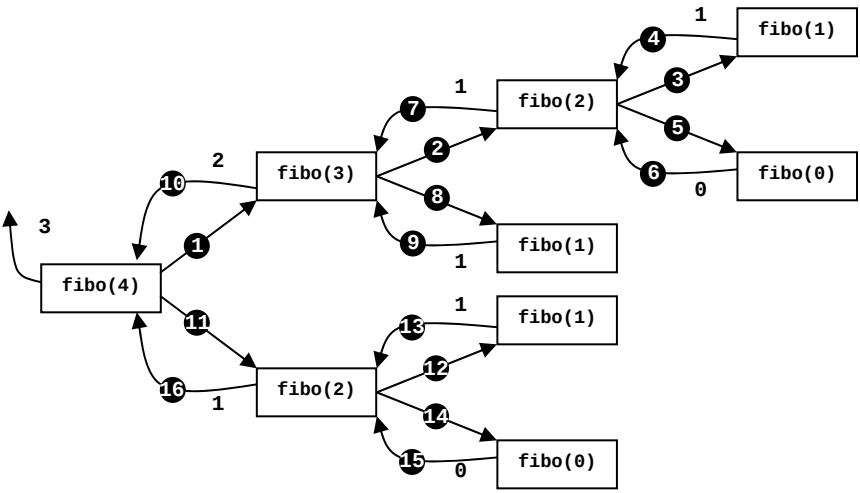


Figure 166: Recursively invoking methods to find fibo(4)

Costs of Recursion

A recursive method accomplishes its task by successively calling itself. Therefore, there are many invocations of method involved. As we have discussed in Chapter 8, the mechanism of method invocation consists of steps such as copying values of the input variables to the local variables of the method, and copying the returned values from inside the method to the caller. These steps make method invocation relatively computationally expensive compared to evaluating expressions and looping through sets of statements using iterative constructs.

Furthermore, each time a recursive call is made, a certain amount of memory must be allocated. For a recursive method that makes very deep recursions, a large amount of memory is required.

Consider *fibo()* in the previous example. We can see that to find **`fibo(4)`**, 8 method invocations are made, and the recursion goes 3 level-deep. For an input value of more than 30, there are more than 1 million method invocations made, and the depth can be more than 30 levels.

Does this mean we should avoid recursive algorithms? No, it does not. Sometimes, the easiest and the least error-prone ways to write programs for solving some problems are recursive methods. Sometimes, an iterative approach is much more difficult than the recursive ones. The examples that we have discussed so far might not serve as a good example to this claim. However, solving the "Towers of Hanoi" problem presented later in this chapter should serve as a convincing example. Try solving it using an iterative approach.

Reducing Numbers of Method Calls

Sometimes we can redesign our recursive methods so that the numbers of times the methods are invoked can be reduced. One trick is to introduce a storage variable, possibly an array variable, which stores the values returned by a method when it is invoked with a set of input values. Then, at any given time that the method is to be invoked in order to be evaluated for its returned value based on its corresponding set of input arguments, the program should check the stored values in the storage variable first and see whether the method with that input has been called and evaluated before or not. If such a value has already been stored in the storage variable, that value can be used right away instead of having to invoke the method again.

The example on Fibonacci numbers are revisited below. This time, we will apply the trick mentioned just above in order to reduce the number of method calls.

Example 86: Fibonacci numbers revisited

Let's revisit the method *fibo()* presented previously. Most of the time, calculating the n^{th} Fibonacci number involves redundant method invocations. For example, from the diagram showing method invocation in **`fibo(4)`**, we can see that **`fibo(2)`** and **`fibo(0)`** are called twice, while **`fibo(1)`** is called three times. An alternative implementation that should save some numbers of method invocation is to introduce a variable storing previously computed values of Fibonacci numbers. If desired

Fibonacci number has been computed earlier, the program should read from the stored values instead of making a method call.

The following program computes the n^{th} Fibonacci number using the method fibo() which is the same as in the previous example, and the method *fiboNew()* in which an array of **int** is used for remembering the Fibonacci numbers that have already been computed. A *println()* statement is added into both *fibo()* and *fiboNew()*, so that it prints a message whenever the method is called. The number of message printed determines how many times each method is called. (To count how many times each method is called, we could introduce static variables used for storing the counts of method invocations. However, static variables have not been discussed until the next chapter.)

```
import java.io.*;                                                    1
public class FiboDemo2                                               2
{                                                                    3
  public static void main(String[] args) throws IOException         4
  {   BufferedReader stdin =                                         5
          new BufferedReader(new InputStreamReader(System.in));      6
      System.out.print("Enter n:");                                  7
      int n = Integer.parseInt(stdin.readLine());                    8
      System.out.println("---Using fibo()-----------");              9
      System.out.println("F("+n+")="+fibo(n));                       10
      System.out.println("---Using fiboNew()---------");             11
      System.out.println("F("+n+")="+fiboNew(n));                    12
  }                                                                  13
                                                                     14
  // The same fibo() as the previous example                         15
  public static int fibo(int n){                                     16
      System.out.println("fibo("+n+") is called.");                  17
      if(n<=0) return 0;                                             18
      if(n==1) return 1;                                             19
      return fibo(n-1)+fibo(n-2);                                    20
  }                                                                  21
                                                                     22
  // The new method in which already computed values                 23
  // are stored in an array of length n+1                            24
  public static int fiboNew(int n){                                  25
      int [] remember = new int[n+1];                                26
      for(int i=0;i<=n;i++) remember[i]=-1;                          27
      return fiboNew(n,remember);                                    28
  }                                                                  29
  public static int fiboNew(int n,int [] r){                         30
      System.out.println("fiboNew("+n+") is called.");               31
      if(n<=0){                                                      32
```

(continued on next page)

(continued from previous page)

```
            r[0]=0;                                                    33
            return r[0];                                               34
        }                                                              35
        if(n==1)                                                       36
            r[n]=1;                                                    37
        else                                                           38
            r[n]=(r[n-1]==-1?fiboNew(n-1,r):r[n-1])                    39
                    + (r[n-2]==-1?fiboNew(n-2,r):r[n-2]);              40
        return r[n];                                                   41
    }                                                                  42
}                                                                      43
```

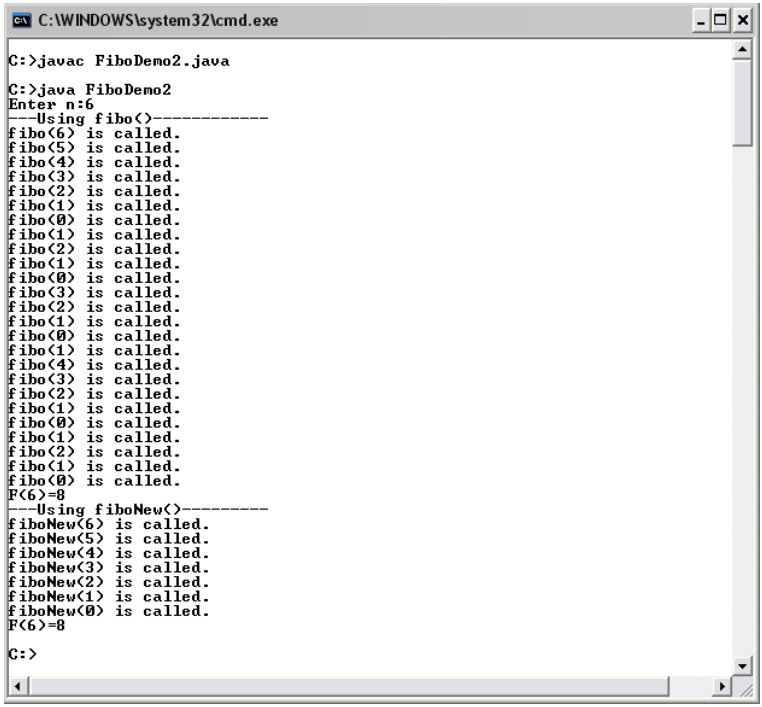


Figure 167: Comparison of invoking fibo() and fiboNew()

From Figure 167, we can see that finding the 6th Fibonacci number using *fibo()* requires more than three times as many method invocations as it is required in the case of using *fiboNew()*.

Example 87: The towers of Hanoi

The Towers of Hanoi is a puzzle invented in the late nineteenth century by the French mathematician Édouard Lucas. The setting of this puzzle consists of three pegs mounted on a board together with disks of different sizes. Initially, these disks are placed on the first peg (peg A) in order of their sizes, with the largest one on the bottom and the smallest one on the top, as shown in the picture below. The goal of this puzzle is to move all disks from their initial locations on Peg A to PegB. However, a valid move in this puzzle must obey two rules:

- Only one disk can be moved at a time, and this disk must be top disk on a tower.
- A larger disk cannot be placed on the top of a smaller disk.

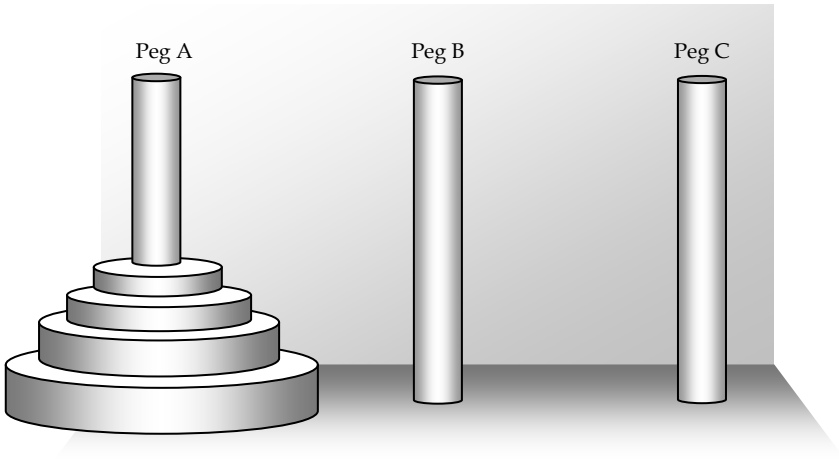


Figure 168: The initial setting of the towers of Hanoi puzzle

Here, we would like to develop a program that finds the moves necessary to complete the puzzle with n disks, where n is a positive integer.

Let's try making the list of the necessary moves for some small values of n.

Starting from n = 1, the only move needed is to move the disk from Peg A to Peg B.

For n = 2, let's call the smaller disk Disk1 and the other disk Disk2. The moves needed are:

- Move Disk1 from Peg A to Peg C.
- Move Disk2 from Peg A to Peg B.
- Move Disk1 from Peg C to Peg B.

For n = 3, the steps needed to complete the puzzle start from performing the moves required to move the top two disk from Peg A to Peg C using a similar set of moves used in solving the puzzle with n=2, but from Peg A to Peg C instead of to the final destination, Peg B. Then, move the biggest disk from Peg A to Peg B. Finally, we can again use the set of moved used in moving two disks from one peg to another peg to move the two disks left on Peg C to Peg B. Therefore, the puzzle is solved for n = 3.

It is easy to notice that to solve the puzzle for any n, the move list starts from the moves required to move the top n-1 disks from one peg to another, which is from Peg A to Peg C, then a move that takes the largest disk from Peg A to Peg B, and, finally, another set of moves required to move n-1 disks from one peg to another which, this time, is from Peg C to Peg B. Therefore, for any n, the problem can be solved recursively, starting from the case where there is only one disk.

Naming the disks from top to bottom with Disk1, Disk2, to Disk$n$, the following Java program, making use of a recursive method, produces the required move list in the following format:

```
Move [disk] from [origin peg] to [destination peg].
```

Here is the program listing.

```
import java.io.*;                                                    1
public class TowerOfHanoiDemo                                        2
{                                                                    3
    public static void main(String[] args) throws IOException       4
```

(continued on next page)

(continued from previous page)

```
    {   BufferedReader stdin =                                          5
            new BufferedReader(new InputStreamReader(System.in));       6
        System.out.print("Enter number of disks:");                     7
        int n = Integer.parseInt(stdin.readLine());                     8
        move(n,"A","B","C");                                            9
    }                                                                   10
    public static void move(int n,                                      11
        String orgPeg,String destPeg,String otherPeg){                 12
        String step;                                                    13
        if(n<=1){                                                       14
            step = "Move Disk1 from Peg ";                              15
            step += orgPeg+" to Peg "+destPeg;                          16
            System.out.println(step);                                   17
        }else{                                                          18
            move(n-1,orgPeg,otherPeg,destPeg);                          19
            step = "Move Disk"+n+" from Peg ";                          20
            step += orgPeg+" to Peg "+destPeg;                          21
            System.out.println(step);                                   22
            move(n-1,otherPeg,destPeg,orgPeg);                          23
        }                                                               24
    }                                                                   25
}                                                                       26
```

In the program, it is amazing in a sense that, apart for statements dealing with the user input, there is only one statement, **`move(n, "A", "B", "C")`**, calling a recursive method that is needed for solving the Towers of Hanoi puzzle, no matter how big the value of **`n`** is.

The *move()* method is defined so that it takes four input arguments. The first one is the number of disks, **`n`**, while the others are the names used to call the peg where all the **`n`** disks are located at the moment where the method is invoked, **`orgPeg`**, the peg which is the destination of the **`n`** disks of interest, **`destPeg`**, and the other peg, **`otherPeg`**.

The base case of the *move()* method is when the number of disk reaches 1. In that case, the move needed is to move that disk from **`orgPeg`** to **`destPeg`**. For the recursive cases with *k* disks, the operation is divided into three subtasks.

The first subtask is to move Disk1 to Disk*k-1* from **`orgPeg`** to **`otherPeg`**. This can be done simply by calling *move()* again but with input parameters being set properly.

The second subtask is to move the bottom-most disk, Disk*k*, from **`orgPeg`** to **`destPeg`**. The last subtask is to move Disk1 to Disk*k-1* from **`otherPeg`** to **`destPeg`**. Again, this last subtask can simply be done by calling *move()* again but with input parameters being set properly.

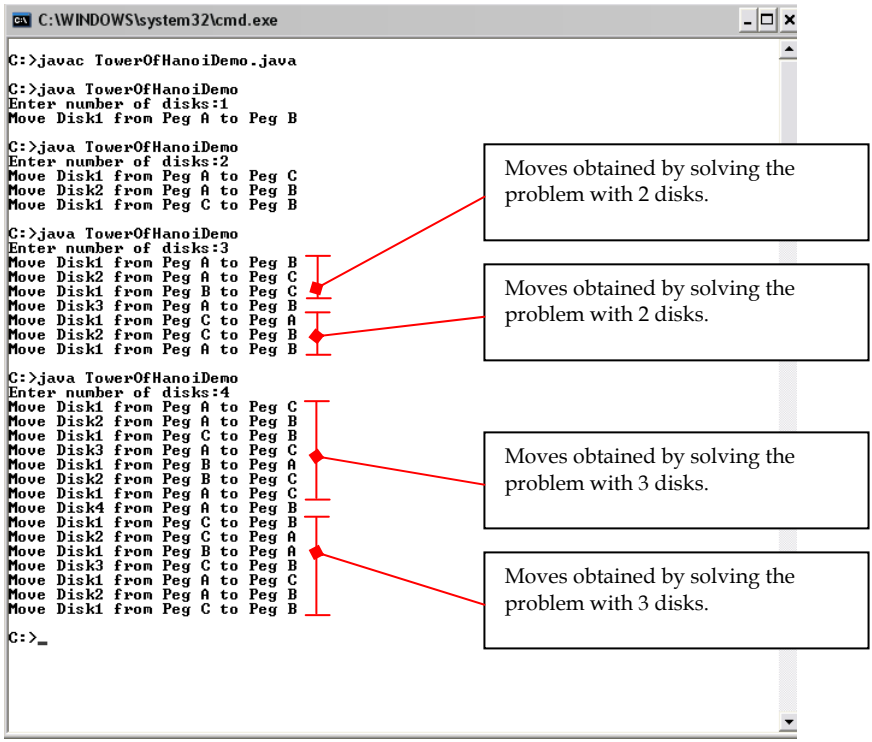


Figure 169: Solutions to the Towers of Hanoi puzzle

Exercise

1. What is the output when *main()* is executed?

```
public static void main(String[] args)
{
        System.out.println(f(5));
}
public static int f(int n){
        if(n<=0) return 1;
        return f(n-1)+2*f(n-2);
}
```

2. What is the output when *main()* is executed?

```
public static void main(String[] args)
{
        System.out.println(g(4));
}

public static int g(int n)
{
        if(n<=0) return 1;
        return 2*g(n-1)+ f(n+1);
}

public static int f(int n)
{
        if(n<=0) return 1;
        if(n==1) return 2;
        return f(n-1)-f(n-2);
}
```

3. Implement the following method using a recursive approach.

```
public static int f(int n)
{
        int a = 0;
        for(int i=1;i<=n;i++){
                a += 2*i;
        }
        return a;
}
```

4. Implement the following method using a recursive approach.

```
public static int f(int n)
{
        int a = 1;
        int b = 1;
        for(int i=1;i<=n;i++){
                a += ++b;
                b = a;
        }
        return a;
}
```

5. Implement the following method using a recursive approach.

```
public static int f(int n)
{
        int a=0,b=1,c=0;
        if(n==1) return b;
        for(int i=2;i<=n;i++){
                a = b+c;
                c = b;
                b = a;
        }
        return a;
}
```

6. Implement the following method using a recursive approach.

```
public static double f(int n)
{
        double a=2.0,b=2.0,c=2.0;
        if(n==1) return b;
        for(int i=2;i<=n;i++){
                a = b+0.5*c+i;
                c = b;
                b = a;
        }
        return a;
}
```

7. Write a Java method for calculating the following function at a given non-negative integer n. Based on your implementation, plot the number of method invocation made as a function of n ranging from 0 to 10.

$$f(n) = \begin{cases} 1 & \text{if } n = 0,1,2 \\ f(n-2) + n \times f(n-3) & \text{if } n = 3,4,... \end{cases}$$

8. Write a Java method for calculating the following function at a given non-negative integer n.

$$f(n) = \begin{cases} \sum_{k=0}^{n} k & ; n = 0,1,2 \\ \frac{1}{2} f(n-1) + \frac{1}{2} f(n-3) & ; 3 \le n < 10 \\ \sum_{k=n-3}^{n-1} (f(k) - (-1)^k f(k-1)) & ; n \ge 10 \end{cases}$$

9. Write a recursive Java method that calculates the sum of every **`int`** element in an input array of **`int`**.

10. Write a recursive Java method that finds the smallest value in an input array of **`int`**.

11. Write a recursive Java method that returns the index of the smallest value in an input array of **`int`**.

12. The mathematical constant e is the base of the natural logarithm. It is called *Euler's number* or *Napier's constant*. This constant is the sum of the infinite series defined below.

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

Write a method to approximate the value of e using the above formula but with a finite number of terms. In other words, instead of having n running to ∞, it runs to an **`int`** value, N, input to the method. Utilize the recursive approach as appropriate. Then, write a Java program to compare the results from the approximation at N = 5, 10, 15, and 20 with the value of **`Math.exp(1)`**.

Note that 0! equals 1. Also, you may use the following method to find the value of n!.

```
public static double factorial(int n)
{
        if(n<=0) return 1;
        return n*factorial(n-1);
}
```

13. Write a recursive method called *printInt()* that receives an **`int`** value as its input, and prints the associated digits on screen one digit on each line, starting from the left. You are not allowed to directly convert the input **`int`** value to *String*. Assume that the input value is always positive. Also draw a picture showing the method invocations occurred when **`printInt(1942)`** is called.

14. The greatest common divisor (gcd) of two integers is the largest integer that divides both of them. For example, the gcd of 74 and 111 is 37.

    a. Write a method that finds the gcd of two input integers using an iterative approach.
    b. Repeat part a. using an recursive approach based on:

$$\gcd(a,b) = \begin{cases} b & \text{if } a\%b = 0 \\ \gcd(b, a\%b) & \text{otherwise} \end{cases}$$

15. A *palindrome* is a word, phrase, number or other sequence of units (such as a strand of DNA) that has the property of reading the same in either direction where the punctuations and spaces are generally ignored. For example, "civic", "level", "Was it a cat I saw?", and "A man, a plan, a canal: Panama" are palindromes.

    Write a recursive Java method that returns **`true`** if the *String* passed to the method is a palindrome. Otherwise, it returns **`false`**.

Chapter 11: Creating Classes

Objectives

Readers should

- Recall the meaning of classes and objects in Java
- Know the components in the definition of a Java class
- Understand how constructors work
- Be able to create class and object methods
- Be able to create new Java classes and use them correctly

Defining Your Own Data Type

Recall that there are two categories of data in Java, namely primitive data type and class. As mentioned earlier, we cannot create a new primitive data type. However, most of the time programmers want to create new data types; they can do so by creating new classes containing *attributes* and *behaviors* of the desired data types. Creating a new class, we write a *class definition* associated with that class in specific Java syntaxes, and save the definition in a separated .java file named after the name of the class. Once the definition is created, other programs can utilize the newly created data type or class in a similar fashion to the primitive data types or other existing classes. Go back and consult Chapter 5 if you cannot recall the meaning of objects and how they are related to classes.

The structure of a Java class definition for a class named **`ClassName`** is:

```
public class ClassName
{
    // Details of the class goes
    // in here.
}
```

This looks like a Java program that we have covered since Chapter 2 but a class definition that is not intended to be executable will not contain the *main()* method. The body of the class definition, i.e. between the pair of

curly braces opened after the name of the class, is the place where variables and methods specific to that class will be.

Example 88: A blank class

Suppose that we would like to create a new data type for representing points in a Cartesian co-ordinate, we could create a new class called *MyPoint*, whose definition follows the following structure.

```
public class MyPoint
{
    // a blank class definition
    // there're no details yet
}
```

This definition has to be saved using the name MyPoint.java. Then, we can write another program that makes use of this class. For example, we could create another program that looks like:

```
public class TestMyPoint1                                    1
{                                                            2
        public static void main(String[] args)              3
        {                                                    4
            MyPoint p, q;                                    5
            p = new MyPoint();                               6
            q = new MyPoint();                               7
        }                                                    8
}                                                            9
```

In the above program, variables **p** and **q** are declared as variables of the type *MyPoint* on line 5 using a similar syntax as when variables of other types are declared. On line 6 and line 7, **p** and **q** are assigned with, or in other words, are made to refer to, new instances, or objects, of the class *MyPoint* using the keyword **new**. This program just shows us a valid way to make use of the newly created class. It has not done anything useful since we did not define anything inside the definition of *MyPoint*.

In reality, we want to put something useful in the definition of our classes so that they are not just blank as it appears in *MyPoint* above.

Java Programs and Java Classes

Notice that source codes of Java programs that we have written so far take the same structure as class definitions that we have just introduced in this chapter. Actually, both Java programs and Java classes are the same things. When an executable program is needed, a Java class with the *main()* method is created and it can be executed with *java.exe*. When a Java class is not intended to be executable but it is intended only for describing a new type of data, there is no need for the *main()* method. However, it should not be surprising to you to run into a Java class that contain the *main()* method as well as a bunch of some other things that describe a new type of data in the same .java file. Some Java classes even create instances of the data type described by itself in its *main()* method.

Components of Class Definitions

The main functionality of a class definition is to define attributes and behaviors of that class. Attributes are entities defining properties of an object. Behaviors are actions (or reactions) of an object. The table below shows example attributes and behaviors of some objects.

Object type	Attributes	Behaviors
Point in a 2D space	The x coordinate The y coordinate etc.	Moving the point a specified location Calculating distance from the point to a specified location etc.
Graphical line in a 3D space	Location of the starting point Location of the ending point Color etc.	Calculating the length of the line Moving the starting point to a specified location Moving the ending point to a specified location Changing the color of the line etc.
Complex number	Value of the real part Value of the imaginary part etc.	Adding the object with another complex object, Multiplying the object with another complex object Finding the conjugate of the object Setting the real part to a specific number Showing String representation of the object

		etc.
Matrix	Members of the matrix etc.	Adding elements to the object Finding determinant Adding the object with another matrix object Finding the inverse of the object Raising to the power of n etc.
Car	Body color Dimensions Weight Number of doors Manufacturer Engine status etc.	Starting the engine Shutting down the engine Showing the name of its manufacturer Accelerating Decelerating etc.
Bank account	Account name Owner Account type Balance etc.	Showing the current balance Showing all info associated with the account Withdrawing money from the account Depositing to the account Closing the account etc.
Customer	Customer ID First name Family name Credit line Gender Favorite products etc.	Showing all info of the customer Changing the credit line Checking whether the customer's favorite product consists of a specified product etc.

Table 11: Examples of real-world objects together with their attributes and behaviors

To describe attributes and behaviors of objects of the class, a class definition can consist of the following components.

1. data members or fields
2. methods
3. constructors

An object's attribute is represented using a *data member*. Variables used for storing data members are called *instance variables*. The behaviors of an object are described using *methods*. *Constructors* are special methods invoked whenever objects of the class are created.

The class definition shown below serves as an example aiming at giving you a very broad overview the structure and syntaxes of a class definition. Details are reserved for later sections.

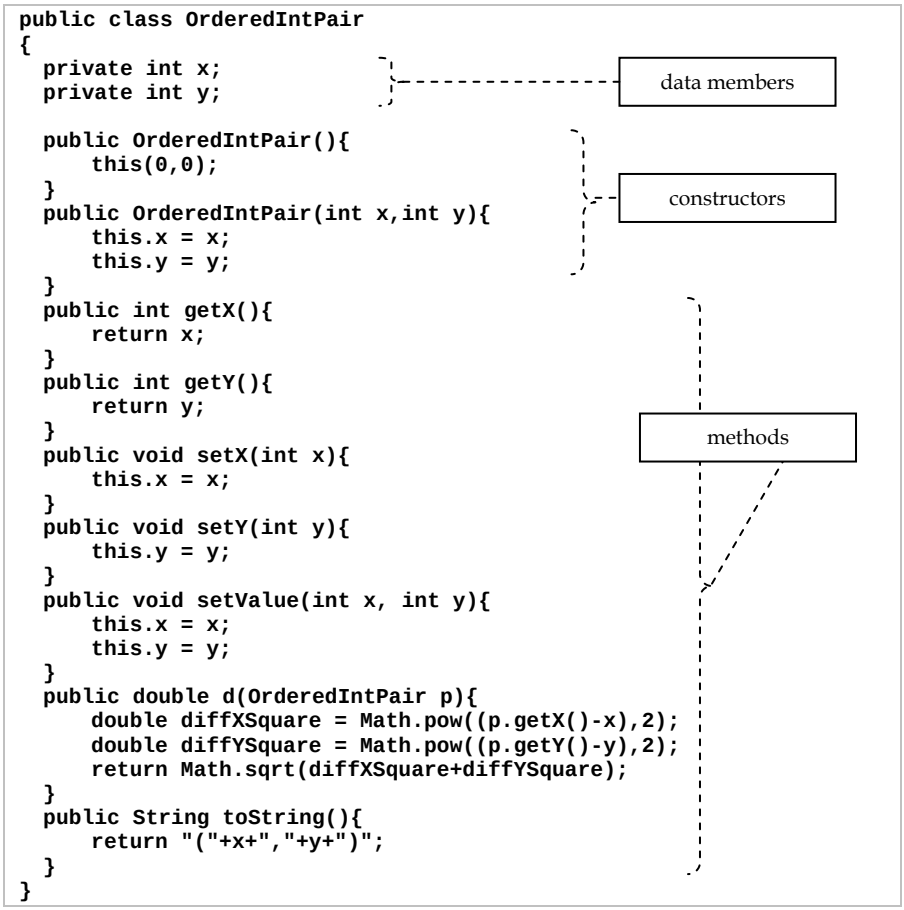


Diagram for Class Descriptions

Starting from chapter 5, we have used some illustrations to describe classes and objects. That we intend to keep those illustrations simple makes those illustrations non-standard (meaning that they are used only

for readers of this book). However, it is good to take a look at how people draw diagram to represent class definitions.

A popular diagram use for describing the details of a class is the *class diagram* defined in the *Unified Modeling Language* (UML). In this diagram, a class is shown using a rectangle in which its name, its data members and its methods (including its constructors) are listed in separated sections in the rectangle. Figure 170 shows a class diagram describing the *OrderedIntPair* class listed above. Java syntaxes are used to describe the class's resources in this diagram.

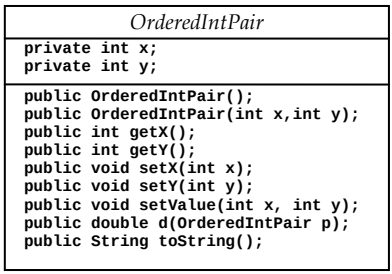


Figure 170: A class diagram

An instance (object) of a class is normally represented using a diagram very similar to the diagram shown in Figure 170 but with its method part omitted, each variable assigned with a value associated with the object, and the object name specified together with the class name. Object diagrams describing three sample objects of the *OrderedIntPair* class are shown in Figure 171. In the figure, there are three objects whose names are **`firstPair`**, **`secondPair`**, and **`thirdPair`**. All of them are instances of the *OrderedIntPair* class. The values of **`x`** and **`y`** for each object is also shown.

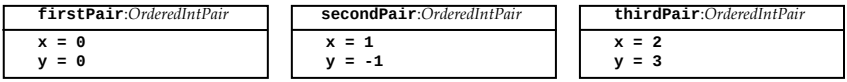


Figure 171: A diagram describing three objects

Readers interested in learning more about UML should consult textbooks on Software Engineering such as [Pre2010] or textbooks dedicated to describing the UML itself such as [Mil2006] and [Pil2005].

Instance Variables and their Access Levels

Instance variables are used for storing data members. Instance variables can be declared and possibly initialized inside a class definition but outside of any methods defined in the class. The syntaxes used for instance variable declaration are similar to the ones using to declare variables inside a method.

Apart from the identifiers of the instance variables used for data members and their data types, another aspect that ones wish to create new data types need to concern about their data members is the specification about their access levels.

Access level modifiers are used when instance variables are declared in order to determine who (or which parts of the running program) have the access rights to those instance variables. These modifiers are **`public`**, **`private`**, and **`protected`**. Either one of these modifier is placed in the instance variable declaration statements before the name of the data type used in the declaration, such as

```
public int k = 1;
private double [] d;
protected String s1;
```

When an instance variable is declared as **`public`**, the dot operator can be used to access the value of this instance variable from any instructions of the running program. In contrary, if an instance variable is declared as **`private`**, it is kept private to the class it is declared. That means the dot operator cannot be used to access its value from anywhere outside the class definition.

Another modifier determining access levels is the modifier **`protected`**. Protected instance variables can only be accessed using the dot operator within the class definition in which the instance variables are declared

and within the class definition of any *subclasses* of the class in which they are declared. Subclasses will be discussed in Chapter 12.

If an instance variable is declared without an explicit access level modifier, its access level is at the default level. In this case, dot operators can be used to access this instance variable from only the definition of its class and classes that are arranged in the same package as its class. The discussion about arranging Java classes in packages is beyond the scope of this book.

Therefore, in this chapter, we will pay attention to only the **`public`** and **`private`** modifiers. Let's notice the effect of access level modifiers in the following examples.

Example 89: Points in the Cartesian coordinate

For an object of the class *MyPoint* to represent a point in the Cartesian coordinate, it should at least have two **`double`** values representing the x-coordinate and the y-coordinate of the point represented by the object. Consequently, the class definition could look like:

```
public class MyPoint
{
        public double x;
        public double y;
}
```

The modifier **`public`** identifies that anyone can access the two instance variables using the dot operator. The following program demonstrates how the values of the two instance variables are accessed.

```
public class TestMyPoint2                                          1
{                                                                  2
        public static void main(String[] args)                    3
        {                                                          4
            MyPoint p = new MyPoint();                             5
            MyPoint q = new MyPoint();                             6
            p.x = 2;                                               7
            p.y = 3;                                               8
            q.x = 0.5;                                             9
            q.y = -0.5;                                            10
```

(continued on next page)

```
(continued from previous page)
            System.out.println("("+p.x+","+p.y+")");                    11
            System.out.println("("+q.x+","+q.y+")");                    12
        }                                                              13
}                                                                      14
```

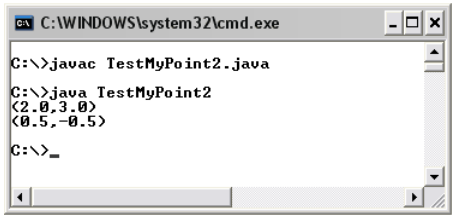


Figure 172: Accessing values of public instance variables

On line 5 and line 6, the variables named **p** and **q** are created. Each of them is made to refer to a new *MyPoint* object. The instance variable **x** of the object referred to by **p** is set to 2 while **y** is set to 3 on line 7 and line 8. On the next two lines, the instance variable **x** of the object referred to by **q** is set to 0.5 while **y** is set to -0.5. The code on line 11 and line 12 print the output on the screen. They use the values of **x** and **y** in both objects through **p.x**, **p.y**, **q.x**, and **q.y**.

Now if we change the class definition of *MyPoint* to:

```
public class MyPoint
{
        private double x;
        private double y;
}
```

Compiling TestMyPoint2.java again will lead to compilation errors as shown in the picture below.

Figure 173: Compilation errors due to attempts in accessing values of private instance variables from outside if their class definition

The modifier **`private`** makes instance variables private to the class they are declared. That means the instance variables can be used or accessed by that class or in the class definition of that class only. Errors occur whenever the private instance variables **`x`** and **`y`** of any instances of *MyPoint* are accessed directly by other classes. In this case, the class trying to access those variables is *TestMyPoint2*. The modifier **`private`** allows the creator of the class to hide data members from the outside world. Doing this is crucial to the *data encapsulation* concept in *Object-Oriented Programming (OOP)*. However, we do not intend to elaborate on OOP concepts in this course. Interested readers can learn more about OOP concepts from many online and printed resources such as [Wei2008] and [Jia2002].

Object Composition

Data members of a class can be objects of other classes which can be either Java standard classes or newly-created classes. For example, let's suppose we would like to create a class *MyLabelledPolygon* representing

polygons, each of which has its associated text label. We might decide that its data members include an array of *MyPoint* objects for storing the location of every vertex of the polygon, and a *String* object representing the label. An instance of this polygon could look like the one in Figure 174.

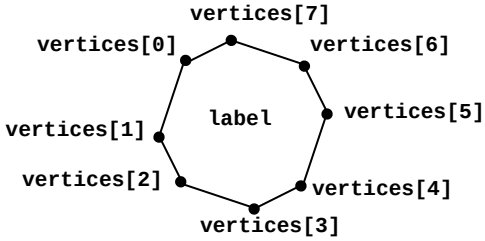


Figure 174: A Polygon instance

The class definition could be listed as the code below.

```
public class MyLabelledPolygon
{   private MyPoint [] vertices;
    private String label;

    // ... other elements are omitted ...
}
```

Since an object of the class *MyLabelledPolygon* is an object that is composed of some *MyPoint* objects and a *String* object, we can say that *MyLabelledPolygon* is a new class that is a composition of objects of other classes. The way to combine objects into an object of a new class can be called *object composition*.

A class created by object composition is said to have the '*has-a* (*has-an*)' or '*has-some*' relationship with the objects it is composed of. For example, in the case of *MyLabelledPolygon*, we can say that an object of the class *MyLabelledPolygon* 'has some' objects of the class *MyPoint* and 'has an' object of the class *String*.

When an object of the class *A* is composed of one or many objects of the class *B*, we can use a UML diagram shown in Figure 175 to indicate their

relationship. In the diagram, the diamond shape at the end of the line linking the two classes indicating the 'has-a' or 'has-some' relationship in which the class near the diamond is the composition of the class at the other end of the line.

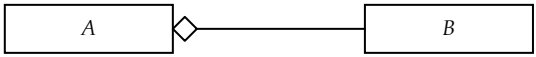


Figure 175: Object composition

The diagram in Figure 176 shows the relationships among *MyLabelledPolygon*, *MyPoint*, and *String*.

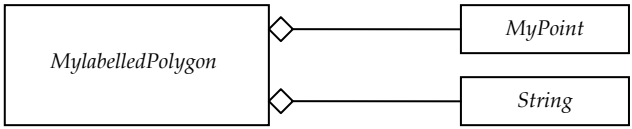


Figure 176: Relationships among MyLabelledPolygon and the classes of its data members.

Static and Non-static Data Members

Data members can be either *static* or *non-static*. Non-static data members are attributes of instances of the class, while static data members are attributes of the class itself. In other words, each instance of the class has its own copy of non-static data members, while static data members are shared among every instances of the class. Data members are non-static by default. To make a data member static, we use the modifier **`static`** in front of the declaration of variables storing the data members. Therefore, to be precise, we will not call variables storing static data members instance variables since the variables are not the attributes of any specific instances but they are shared among every instances. Such variables may be called *class variables*.

Example 90: Static vs. non-static data members

The following program shows how static and non-static variables are declared and used.

```
public class C11A                                                    1
{                                                                    2
        public static int i;                                         3
        public int j;                                                4
}                                                                    5
```

```
public class StaticDataMemberDemo                                    1
{                                                                    2
        public static void main(String[] args)                      3
        {                                                            4
            C11A x = new C11A();                                     5
            C11A y = new C11A();                                     6
            C11A z = new C11A();                                     7
            x.j = 5;                                                 8
            y.j = 10;                                                9
            z.j = 15;                                                10
            System.out.println("x.j = "+x.j);                        11
            System.out.println("y.j = "+y.j);                        12
            System.out.println("z.j = "+z.j);                        13
            x.i = 0;                                                 14
            y.i++;                                                   15
            z.i += 3;                                                16
            System.out.println("x.i = "+x.i);                        17
            System.out.println("y.i = "+y.i);                        18
            System.out.println("z.i = "+z.i);                        19
        }                                                            20
}                                                                    21
```

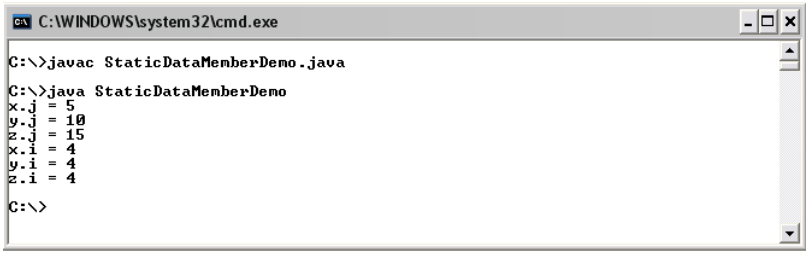


Figure 177: Demonstration of using static and non-static data member

On line 5, line 6, and line 7, three instances of *C11A* are created and referred to by **x**, **y** and **z**. On line 8, line 9, and line 10, the values of 5, 10,

and 15 are assigned to the instance variables `j` belonging to the objects referred to by `x`, `y`, and `z`, respectively. These objects do not share the value of `j`. However, the variable `i` is shared by the three objects. The statement `x.i = 0` on line 14 assign 0 to `i`. Note that at this point `y.i` and `z.i` are also 0 since they refer to the same thing. `i` can be modified via any objects. Therefore, we can see that the resulting value of `i`, shared by `x`, `y` and `z`, is 4.

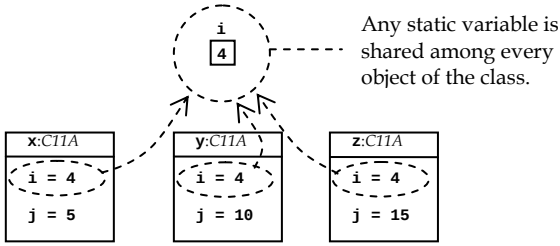


Figure 178: A diagram showing three objects of the C11A class and the values of their instance and class variables

---

Methods

Methods describe behaviors of objects of the class. We have learned how to use methods defined in existing classes in Chapter 5. In Chapter 5, we also mentioned that there were two types of methods: static (class) methods and non-static (instance) methods. A (public) method defined in a class definition is by default non-static and it can be invoked by other classes via the instance name of an object of that class using the dot operator. To make a method static, the keyword *static* is put in the method header. This way, the method can be invoked using the dot operator with the name of the class. The general syntax of defining a method in a class is similar to what we have already been familiar with in Chapter 8. This time, we will look at the syntax in a more general view. The syntax follows:

```
(modifier) (static) returnType methodName(argumentList){
       methodBody
}
```

An access level modifier (shown as **(modifier)**), being either **public, private**, or **protected**, can be specified at the beginning of the method header. It determines whether which classes can make use of this method. The access levels specified by **public, private**, and **protected** are similar to when they are used with data members.

The keyword **static** makes the method static, or a class method. If omitted, the method is considered as non-static, or an instance method.

The other parts of the method definition are the same as what we discussed in Chapter 8.

We can define as many methods as we would like in the class definition. If the definition contains a public method named *main()*, the class can be executed by *java.exe*. In other words, the class is in fact a Java program.

Discriminating data Members and Variables Declared inside Methods

When declaring a method inside a class, it is okay to name the variables inside the input argument list of the method with the identifier similar to a variable used for a data member of the class. However, since both data members and variables declared in the method can be used inside the method, programmers need to know how to distinguish between variables for data members and variables declared in the method with the same identifiers. Situations in which programmers choose to use ambiguous variable names are not uncommon at all. Some examples of such situations can be seen in accessor and mutator methods which will be discussed shortly after this.

The **this** keyword is used with the dot operator for this situation. Suppose that there is a data member that is represented by a variable **var**. In the same class, there is a method whose input argument list contains another variable that is also named **var**. The expression **this.var** will refer to **var** that is the data member, while using **var** without the **this** keyword will refer to **var** that is declared inside the method.

When the variable names are not ambiguous, using the name of the variable alone can mean either the data member or the variable declared in the input argument list, whichever applies. However, using the variable with the **`this`** keyword always means the data member.

Accessor and Mutator Methods

Typically, in OOP, data members in a class are defined as **`private`** to prevent users of the class accessing the data members directly. Instead, the creator of the class usually provides public methods for reading or changing some data members. Methods provided for other classes to read the values of data members are called *accessor methods*, while methods provided for changing the values of data members are called *mutator methods*.

toString()

Whenever an object of a class needs to be converted to its *String* representation, Java automatically calls a specific method called *toString()*. Therefore, in order to provide a meaningful *String* representation of the class we create, it is sensible to provide the method named exactly as *toString()* that returns the *String* representation we want.

Example 91: Accessor, mutator, toString() and other methods

The following code shows a more complex class definition of *MyPoint*. The definition provide appropriate methods, including mutator methods, accessor methods, *toString()* as well as some other useful methods.

```
public class MyPoint                                                    1
{                                                                       2
        // data members                                                 3
        private double x;                                               4
        private double y;                                               5
```

(continued on next page)

(continued from previous page)

```
                                                                    6
        // accessor methods                                         7
        public double getX(){                                       8
            return x;                                               9
        }                                                           10
        public double getY(){                                       11
            return y;                                               12
        }                                                           13
                                                                    14
        // mutator methods                                          15
        public void setX(double x){                                 16
            this.x = x;                                             17
        }                                                           18
        public void setY(double y){                                 19
            this.y = y;                                             20
        }                                                           21
                                                                    22
        // other methods                                            23
        public void setLocation(double x, double y){                24
            this.x = x;                                             25
            this.y = y;                                             26
        }                                                           27
        public double distanceTo(MyPoint p){                        28
            double diffXSquare = Math.pow((p.getX()-x),2);          29
            double diffYSquare = Math.pow((p.getY()-y),2);          30
            return Math.sqrt(diffXSquare+diffYSquare);              31
        }                                                           32
        public String toString(){                                   33
            return "("+x+","+y+")";                                 34
        }                                                           35
}                                                                   36
```

The methods *getX()* and *getY()* declared on line 8 and line 11 allows other classes to read the values of the private variables **x** and **y**, respectively. These are accessor methods. The methods *setX()* and *setY()* declared on line 16 and line 19 allows other classes to set the values of the private variables **x** and **y**. These are mutator methods.

You should notice the usage of **this**. **this** is a reference used for referring to the current instance of the class. On line 17 and line 20, **this.x** and **this.y** refer to the instance variables **x** and **y** of the current instance, i.e. the instance from which the methods are invoked.

Now, let's use this class. Observe the following program and its output.

```
public class TestMyPoint3                                                    1
{                                                                            2
        public static void main(String[] args)                              3
        {                                                                    4
            MyPoint p = new MyPoint();                                       5
            MyPoint q = new MyPoint();                                       6
            p.setX(6.0);                                                     7
            p.setY(5.0);                                                     8
            q.setLocation(p.getX(),p.getY());                                9
            System.out.println("q="+q);                                      10
            p.setLocation(10.0,2.0);                                         11
            System.out.print("Distance from "+p+" to ");                     12
            System.out.println(q+" is "+p.distanceTo(q));                    13
        }                                                                    14
}                                                                            15
```

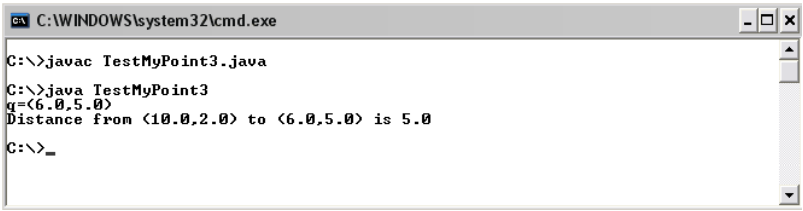


Figure 179: A program using various methods of the *MyPoint* class

On line 7, *setX()* is invoked from **p**. This set the value of **x** belonging to the *MyPoint* object referred to by **p** to the value input to the method. Similarly, the value of **y** belonging to the *MyPoint* object referred to by **p** is set to 5.0 on line 8.

On line 9, **p.getX()** and **p.getY()** return the value of the instance variables **x** and **y** belonging to the *MyPoint* object referred to by **p**. These values are used as input parameters to *setLocation()* invoked from **q**, in which the instance variable **x** of the object referred to by **q** is assigned with the first input parameter to the method, while the instance variable **y** of the object referred to by **q** is assigned with the other input parameter.

Whenever the *String* representation of a *MyPoint* object is needed, for example in argument lists of *print()* and *println()* on line 10, line 12, and line 13, *toString()* of that object is automatically invoked.

Example 92: Utility class with only static methods

Static methods are also useful when we would like to build a class providing useful functionalities to be used by other classes or programs, such as the standard *Math* class. Such a class is not commonly instantiated, or in other words, it is not common to create an object of such a class. Therefore, the functionalities are provided through its public static methods.

Here is a sample class made up for providing some functionality for **`int`** array manipulations. Note that this class serves as an example when static methods are used. There are some smarter ways to manipulate arrays.

```
public class MyIntArrayUtil                                          1
{                                                                    2
  public static int [] createRandomElements(                        3
                                     int n,int min, int max)         4
  {                                                                  5
      int [] a = new int[n];                                         6
      for(int i=0;i<n;i++){                                          7
          a[i] = (int)Math.round(Math.random()*(max-min)+min);       8
      }                                                              9
      return a;                                                      10
  }                                                                  11
  public static void showElements(int [] a)                         12
  {                                                                  13
      System.out.print("[ "+a[0]);                                  14
      for(int i=1;i<a.length;i++){                                   15
          System.out.print(", "+a[i]);                               16
      }                                                              17
      System.out.print(" ]\n");                                      18
  }                                                                  19
  public static int [] removeAt(int [] a,int n){                    20
      if(n<0 || n>a.length-1) return a;                              21
      int [] b = new int[a.length-1];                                22
      for(int i=0;i<n;i++){                                          23
          b[i] = a[i];                                               24
      }                                                              25
      for(int i=n+1;i<a.length;i++){                                 26
          b[i-1] = a[i];                                             27
      }                                                              28
      return b;                                                      29
  }                                                                  30
  public static int [] insertAt(int [] a, int n, int k){            31
      if(n<0 || n>a.length) return a;                                32
```

(continued on next page)

(continued from previous page)

```
      int [] b = new int[a.length+1];                                  33
      for(int i=0;i<n;i++){                                            34
          b[i] = a[i];                                                 35
      }                                                                36
      b[n] = k;                                                        37
      for(int i=n;i<a.length;i++){                                     38
          b[i+1] = a[i];                                               39
      }                                                                40
      return b;                                                        41
  }                                                                    42
}                                                                      43
```

The class *MyIntArrayUtil* created here contains four public static methods. The first one defined on line 3 creates an **int** array of length **n** whose elements are integer randomly chosen from **min** to **max**, inclusively. The method defined on line 12 prints all elements of the input array on screen. The method defined on line 20 removes the element at a specified position. Defined on line 31, the method inserts a given value to a specified position of the input array.

The following program makes use of the public static methods in *MyIntArrayUtil*. Observe the output of the program by yourself.

```
public class TestMyIntArrayUtil                                        1
{                                                                      2
    public static void main(String[] args)                           3
    {                                                                  4
        System.out.print("\nOriginal array:\t\t");                     5
        int [] a = MyIntArrayUtil.createRandomElements(5,1,10);        6
        MyIntArrayUtil.showElements(a);                                7
        System.out.print("insert 6 at 0:\t\t");                       8
        a = MyIntArrayUtil.insertAt(a,0,6);                            9
        MyIntArrayUtil.showElements(a);                                10
        System.out.print("insert 9 at 3:\t\t");                       11
        a = MyIntArrayUtil.insertAt(a,3,9);                            12
        MyIntArrayUtil.showElements(a);                                13
        System.out.print("insert 1 after:\t\t");                      14
        a = MyIntArrayUtil.insertAt(a,a.length,1);                     15
        MyIntArrayUtil.showElements(a);                                16
        System.out.print("remove at 2:\t\t");                         17
        a = MyIntArrayUtil.removeAt(a,2);                              18
        MyIntArrayUtil.showElements(a);                                19
        System.out.print("remove at 0:\t\t");                         20
        a = MyIntArrayUtil.removeAt(a,0);                              21
        MyIntArrayUtil.showElements(a);                                22
        System.out.print("remove the last:\t");                       23
```

(continued on next page)

(continued from previous page)

```
        a = MyIntArrayUtil.removeAt(a,a.length-1);                24
        MyIntArrayUtil.showElements(a);                           25
    }                                                             26
}                                                                 27
```

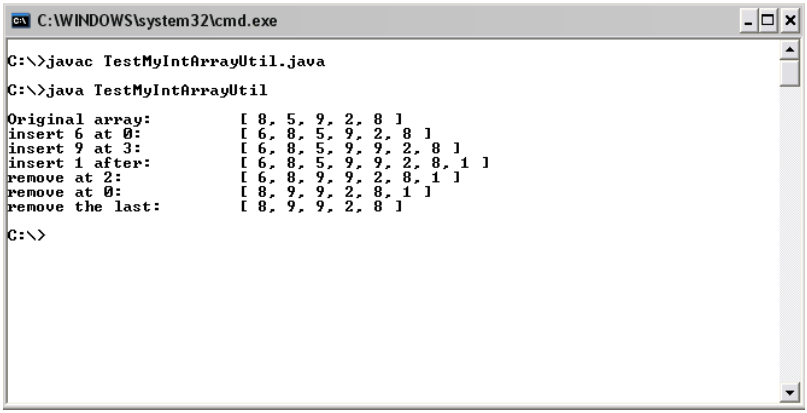


Figure 180: Using static methods inside a utility class

Putting main() into Class Definitions

As mentioned earlier, a Java class definition can also contain the *main()* method. If that is the case, the class can be executed as a Java program and the instructions executed are ones inside the *main()* method. This fact comes in handy when we would like to create a new class and would like to write some instructions to test the usage of the class. To do that, we listed data members as well as methods as usual, but we can also add the *main()* method in which instances of the class can be created and used just like it is done in other programs.

Example 93: Executable class definition

Consider the following class definition that contains the *main()* method.

```
public class ExecutableClass {                                          1
    private int i = 0;                                                  2
    public void setI(int i){                                            3
        this.i = i;                                                     4
    }                                                                   5
    public String doAction(){                                           6
        return "<<"+i+">>";                                             7
    }                                                                   8
    public static void main(String [] args){                            9
        ExecutableClass ex1 = new ExecutableClass();                    10
        ex1.setI(99);                                                   11
        ExecutableClass ex2 = new ExecutableClass();                    12
        ex2.setI(108);                                                  13
        System.out.println(ex1.doAction());                             14
        System.out.println(ex2.doAction());                             15
    }                                                                   16
}                                                                       17
```

In the main() method, the program creates two instances of the *ExecutableClass* class, whose details are listed in the same class as the program. The instance variable i of the first instance is set to 99 by the statement on line 11 and the one of the second instance is set to 108 by the statement on line 13. *doAction()* is called on each instances and the results are printed out onto the screen.

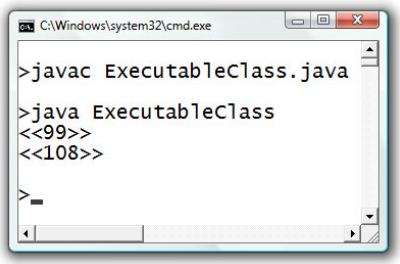


Figure 181: A program creating instances of its own class

Constructors

Constructors are special methods invoked whenever an object of the class is created. Constructors are defined in the same fashion as defining methods. However, constructors must have the same name as the class name and there must not be any return types specified at the header of the constructors. Constructors are usually for initializing or setting instance variables in that class. How they are set is described in the body of the constructor together with other possible intructions.

Given a class called *MyClass*, its constructors are in the following structure.

```
public Myclass(<input argument list>){
    // Body of the constructor
}
```

Here is an example of a no-argument (no input) constructor for *MyPoint*, in which its instance variables are set with 1.0.

```
public MyPoint(){
    x = 1.0;
    y = 1.0;
}
```

Adding this constructor to the class definition of *MyPoint*, we obtain:

```
public class MyPoint
{
    // data members
    private double x;
    private double y;

    // constuctors
    public MyPoint(){
            x = 1.0;
            y = 1.0;
    }

    // …… Details are omitted……

    public String toString(){
            return "("+x+","+y+")";
    }
}
```

Once *MyPoint* is defined this way, whenever a *MyPoint* object is instantiated with `new MyPoint()`, a new object is created with `x` and `y` initialized with `1.0` due to that two statements in the constructor.

Overloading Constructors

Constructors can be overloaded just like methods. A class can have multiple constructors with different input argument lists. Which constructor to be called when an instance of the class is created depends on the input argument list used with the `new` statement.

No-argument Constructor

The no-argument constructor is the constructor that does not take any input arguments. Therefore, it usually contains a set of instructions that provide a default initialization to the object's data members.

Detailed Constructor

The detailed constructor usually refers to the constructor each input argument of which is corresponding to a data member of the class. Typical implementation of this constructor is to initialize every data member with its corresponding input argument.

Copy Constructor

The copy constructor usually refers to the constructor that takes another object of the same class as its input argument. It usually initializes each data member of the new object with the value of the corresponding data member of the input object. This results in that the new object has all of its attributes copied from the original one.

There can be other constructors apart from the three listed above. The rules of overloading constructors are the same as the ones governing the overloading of any other methods.

Example 94: Overloaded constructors

Consider a new class definition of *MyPoint* listed below when overloaded constructors are added.

```
public class MyPoint
{
    // data members
    private double x;
    private double y;

    // constructors
    public MyPoint(){
        x = 1.0;
        y = 1.0;
        System.out.println("MyPoint() is called.");
    }
    public MyPoint(double x,double y){
        this.x = x;
        this.y = y;
        System.out.println("MyPoint(double,double) is called.");
    }
    public MyPoint(MyPoint p){
        x = p.getX();
        y = p.getY();
        System.out.println("MyPoint(MyPoint) is called.");
    }

    // …… Details are omitted……

    public String toString(){
        return "("+x+","+y+")";
    }
}
```

The first constructor, **`MyPoint()`**, does not take any input arguments. Therefore, it is called via the statement **`new Mypoint()`**. Such a constructor is the no-argument constructor. **`MyPoint(double x, double y)`** is a constructor that takes two **`double`** values as its input. It is called via the statement **`new Mypoint(a,b)`**, where **`a`** and **`b`** are any double values. This constructor initializes the instance variables to the input values. Such a constructor that requires the values of the instance variables as its input is the detailed constructor. The last constructor is **`MyPoint(MyPoint q).`** This constructor is invoked as a response to the statement **`new Mypoint(c)`**, where c is an instance of *MyPoint*. In this constructor the value of **`x`** is set to the value of **`x`** from the instance of *MyPoint* supplied as

the input to the constructor, and the value of **y** is set to the value of **y** from the same instance. Such a constructor that copies all attributes from the input instance is the copy constructor. Just like method overloading, you should notice that constructors are not limited to the ones shown in this example. Also note that we add an invocation of *println()* inside each of the constructor to observe that which one of the constructors is invoked when each instance is created.

Observe the output of the following program by yourself. Pay attention to the order of constructors invoked through the messages printed out on the screen.

```
public class TestMyPoint5                                    1
{                                                            2
        public static void main(String[] args)              3
        {                                                    4
            MyPoint p = new MyPoint();                       5
            System.out.println("p-->"+p);                    6
            MyPoint q = new MyPoint(2.0,5.0);                7
            System.out.println("q-->"+q);                    8
            MyPoint r = new MyPoint(q);                      9
            System.out.println("r-->"+r);                    10
        }                                                    11
}                                                            12
```

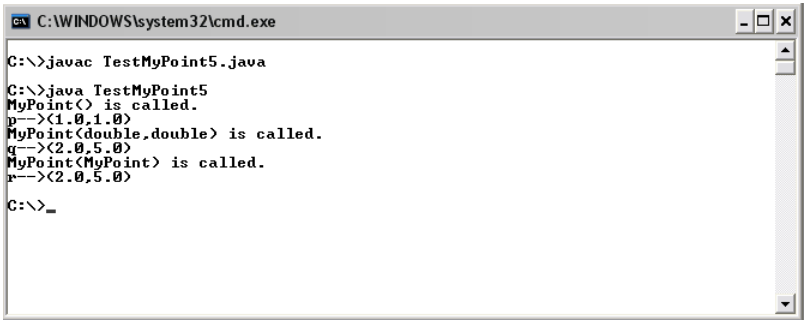


Figure 182: Demonstration of using overloaded constructors

When there is no constructor provided in a class, it is still okay to create an instance of that class using the new statement without any input

arguments. Java can handle the instantiation properly. If the variables for data members are not explicitly initialized when they are declared inside the class definition, default values will be used for those variables based on their data types (zero for numeric data type, **`false`** for **`boolean`**, and **`null`** for non-primitive types). Otherwise, the values explicitly used in the initialization are used.

However, if there is at least one constructor defined in the class other than the no-argument constructor which is absent, Java will treat the new statement without any input arguments in the same way as in the case of other missing overloaded constructors. That means it will produce a compilation error.

Consider the following example.

Example 95: Missing no-argument constructor

The following class definition does not contain the no-argument constructor but it does provide a detailed constructor. In the *main()* method, the program tries to create an instance of this class with the **`new`** statement without any input arguments on line 7. The program cannot be compiled successfully since it cannot find the constructor that does not take any arguments. Java will not instantiate the object using the default mechanism due to the presence of a constructor (which is the detailed constructor, in this case) in the class definition. The error can be observed in Figure 183.

```
public class MissingConstructor {                                    1
    private double d;                                                2
    public MissingConstructor(double d){                             3
        this.d = d;                                                  4
    }                                                                5
    public static void main(String [] args){                        6
        MissingConstructor mc = new MissingConstructor();            7
    }                                                                8
}                                                                    9
```

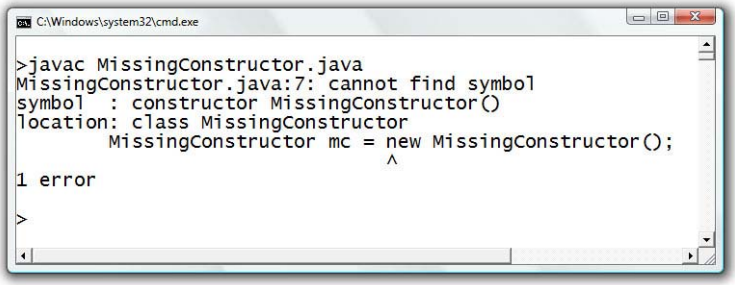


Figure 183: Compilation error due to missing a constructor

Calling a Constructor from Other Constructors

We have mentioned that the detailed constructor is the constructor that assigns values to each data member one by one based on the input supplied to the constructor. Comparing the no-argument constructor with the detailed constructor, we can see that the operation performed by the no-argument constructor can be considered a special case of the operation performed by the detailed constructor. This special case also assigns values to every data members but with a default set of values. The same thing applies when we compare the operation of the copy constructor with the operation of the detailed constructor. The copy constructor assigns values that are fixed by the input object to every data members. This is considered a special case to the operation of the detailed constructor as well. If we consider other possible constructors that attempt to assign values to every data members based on some inputs, their operations can also be considered special cases to the one of the detailed constructor as well.

Therefore, it is usually considered desirable to have such constructors prepare sets of values to be assigned to the data members and make use of the detailed constructor to perform the actual assignment just like what is shown in Figure 184 in the case of the *MyPoint* class.

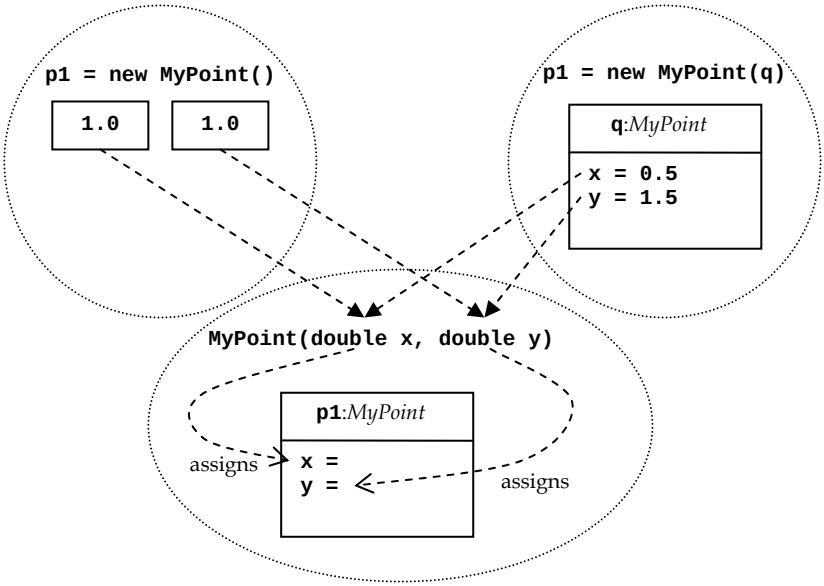


Figure 184: Re-using the detailed constructor

This is considered a good strategy especially when there are some changes made to the data members such as when we decide to change the names of the variables representing the class's data members. If the actual assignments made to the variables are all implemented in the detailed constructor, it is the only place where the source code needs to be changed.

The above paragraphs describe an example situation when there must be a way to invoke a constructor from other constructors. In Java, a constructor can be invoked within another constructor using *this(***`[argument list]`***)*, where `[argument list]` is the list of arguments corresponding to the argument list of the constructor to be called.

There is a limitation that you need to keep in mind. If the invocation of a constructor via *this()* statement is used, the statement must be the first statement in the constructor. Otherwise, it will lead to a compilation error.

Adopting the mention strategy, we can re-implement the constructors of *MyPoint* as listed below.

```
// constructors
public MyPoint(){
        this(1.0,1.0);
}
public MyPoint(double x,double y){
        this.x = x;
        this.y = y;
}
public MyPoint(MyPoint p){
        this(p.getX(),p.getY());
}
```

Notice that if somehow we decide to change the names of the instance variables from **x** and **y** to something else, the only statements we need to change in the two statements inside the detailed constructor. Readers should figure about what would happen in the case of this change if the constructors make actual assignments to the two instance variables instead of calling the detailed constructor to make the actual assignments.

With the availability of the accessor methods, *setX()* and *setY()*, it will be even better if the accessor methods are called by the detailed constructor instead of having the constructor makes the actual assignments by itself. The code for the detailed constructor could look like the following.

```
public MyPoint(double x,double y){
        setX(x);
        setY(y);
}
```

Consider the following non-trivial example where a new data type is defined to represent complex numbers. The class provides useful non-static methods for manipulating the object of this class.

Example 96: Complex numbers

A complex number is of the form $a+jb$, where a and b are real numbers, and j is a quantity representing $\sqrt{-1}$. We would like to define a new class for complex numbers. Complex numbers are added, subtracted,

and multiplied by formally applying the associative, commutative and distributive laws of algebra, together with the equation $j^2 = -1$. Therefore,

$$\begin{aligned}(a+jb)+(c+jd) &= (a+c)+j(b+d)\\ (a+jb)-(c+jd) &= (a-c)+j(b-d)\\ (a+jb)(c+jd) &= (ac-bd)+j(bc+ad)\end{aligned}.$$

The reciprocal or multiplicative inverse of a complex number can be written as:

$$(a+jb)^{-1} = \left(\frac{a}{a^2+b^2}\right) + j\left(\frac{-b}{a^2+b^2}\right),$$

when the complex number is non-zero.

Division between two complex numbers is defined as:

$$\frac{(a+jb)}{(c+jd)} = (a+jb)(c+jd)^{-1}.$$

Complex conjugate of a complex number $a+jb$ is $a-jb$, while the magnitude of $a+jb$ is calculated by $\sqrt{(a^2+b^2)}$.

Here is an example of the class definition for *Complex*, a class we use for representing complex numbers.

```
public class Complex
{
        // attributes: (re) + j(im)
        private double re;
        private double im;

        // constructors
        public Complex(){
            this(0,0);
        }
```

(continued on next page)

(continued from previous page)

```
    public Complex(double r, double i){
        setRe(r);
        setIm(i);
    }
    public Complex(Complex z){
        this(z.getRe(),z.getIm());
    }

    //accessor methods
    public double getRe(){
        return re;
    }
    public double getIm(){
        return im;
    }

    //mutator methods
    public void setRe(double r){
        re = r;
    }
    public void setIm(double i){
        im = i;
    }

    //other methods
    public Complex adds(Complex z){
        return new Complex(re+z.getRe(),im+z.getIm());
    }
    public Complex subtracts(Complex z){
        return new Complex(re-z.getRe(),im-z.getIm());
    }
    public Complex multiplies(Complex z){
        double r = re*z.getRe()-im*z.getIm();
        double i = im*z.getRe()+re*z.getIm();
        return new Complex(r,i);
    }
    public Complex divides(Complex z){
        return this.multiplies(z.multInverse());
    }
    public Complex multInverse(){
        double den = Math.pow(this.magnitude(),2);
        return new Complex(re/den,-im/den);
    }
    public Complex conjugate(){
        return new Complex(re,-im);
    }
    public double magnitude(){
        return Math.sqrt(re*re+im*im);
    }
```

(continued on next page)

(continued from previous page)

```
        public String toString(){
            if(im>=0)
                return re+"+j"+im;
            else
                return re+"-j"+(-im);
        }
}
```

The following program shows the class *Complex* in action. Note that even though we can put this program inside the class definition of *Complex*, it is more realistic to create a separate program as an example of how other people who do not have access to the source code of the class can make use of the class.

```
public class TestComplex                                              1
{                                                                     2
        public static void main(String[] args)                       3
        {                                                             4
            Complex p = new Complex(1,1);                             5
            Complex q = new Complex(3,4);                             6
            System.out.println("p="+p+", q="+q);                      7
            System.out.println("p+q="+p.adds(q));                     8
            System.out.println("p-q="+p.subtracts(q));                9
            System.out.println("p*q="+p.multiplies(q));               10
            System.out.println("p/q="+p.divides(q));                  11
            System.out.println("conjugate of p="+p.conjugate());      12
            System.out.println("magnitude of q="+q.magnitude());      13
        }                                                             14
}                                                                     15
```

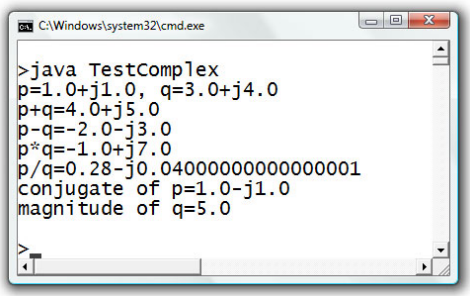


Figure 185: A program testing a data type presenting complex numbers

Exercise

1. Explain the following words:
    a. class
    b. object
    c. attribute
    d. behavior
    e. instance variable
    f. constructor
    g. accessor method
    h. mutator method

2. Identify data members and methods of the following class.

```
public class Ex11_2{
        public double d;
        public Color k;
        public void makeItem(){
        }
        public String describe(){
                return "Ex11_2:"+d+","+k;
        }
}
```

3. Can the following class definition be compiled successfully? If not, explain what is wrong.

```
public class Ex11_3{
        public d;
        public k;
}
```

4. Explain the difference between *static* and *non-static* data members.

5. Explain the difference between *static* and *non-static* methods.

6. Consider the following class definition.

```
public class Ex11_6
{
        public int a,b;
        public int c = 2;
        public static int x = 6;
}
```

   What are the values of **a**, **b**, **c**, and **x**, of **instanceA** and **instanceB** after the following program is run?

```
public class Ex11_6Test
{
        public static void main(String[] args)
        {
                Ex11_6 instanceA = new Ex11_6();
                Ex11_6 instanceB = new Ex11_6();
                instanceA.a = 8;
                instanceB.b = instanceA.x;
                instanceA.x++;
                instanceB.a = 10;
                instanceB.c = 90;
                instanceB.x++;
        }
}
```

7. A file named *Capsule.java* has the following content.

```
public class Capsule
{
        public static int nCapsules = 0;
        public double volume;
        public String screenText;

        public Capsule(double volume,String s){
                this.volume = volume;
                screenText = s;
                nCapsules++;
        }
}
```

Determine the value of **nCapsules** when the following program is run.

```
public class Ex11_7Test
{
    public static void main(String[] args)
    {   int [] nInPack = {5,10,10};
        Capsule [][] pack = new Capsule[3][];
        for(int i=0;i<pack.length;i++){
            pack[i] = new Capsule[nInPack[i]];
            for(int j=0;j<pack[i].length;j++){
                pack[i][j] = new Capsule(0.5,"Formular"+i+j);
            }
        }
        System.out.println(Capsule.nCapsules);
    }
}
```

8. Determine the output of the following program.

```
public class Ex11_8Test
{
        public static void main(String[] args)
        {       int k = 5, j = 6;
                Ex11_8 a = new Ex11_8();
                Ex11_8 b = new Ex11_8(k*j);
                Ex11_8 c = new Ex11_8(k,j);
        }
}
```

   Given that the class *Ex11_8* is defined as:

```
public class Ex11_8
{
    public Ex11_8(){
        this(0);
        System.out.println("A");
    }
    public Ex11_8(int k){
        this(0,0);
        System.out.println("B");
    }
    public Ex11_8(int k,int m){
        System.out.println("C");
    }
}
```

9. Modify the class *MyIntArrayUtil* so that it contains another static method that takes an input array of **int** and sort it in increasing order. Then write a Java program that performs the following steps.

   a. Create an array of ten **int** values, where each value is randomly chosen from 1 to 100.
   b. Show the elements of the array on screen.
   c. Sort the elements of the array increasingly.
   d. Show the elements of the array on screen again.

10. Create a class called *Wrestler* for representing a wrestler in a wrestling simulation program, which simulates the result of a wrestling match with two wrestler based on their wrestling attributes.

    A wrestler has the following attributes.

Attribute Name	Description	Value
Name	The name of the wrestler	a text
Power(pow)	Wrestlers cannot fight when their power reach 0.	a number less than Max. power
Max. power	The power at the starting of a match	100
Strength (str)	The fighting strength of the wrestler	an integer from 1 to 20
Toughness (tou)	The ability to endure damage	an integer from 1 to 20
Stamina (sta)	A wrestler with high stamina can fight longer.	an integer from 1 to 20
Speed (spd)	The agility of the wrestler	an integer from 1 to 20
Luck (lck)	Luck determines how lucky the wrestler is.	an integer from 1 to 20
Courage (cou)	A wrestler with high courage tends to become boosted more often.	an integer from 1 to 20
Skill (skl)	A wrestler with high skill can turn defense to attack more frequently.	an integer from 1 to 20

Wrestler must have a constructor that sets the variable **`name`** to a given *String*. All data members of the class must be **`private`**.

Also provide appropriate mutator and accessor methods. All accessor methods are **`public`**. However, for **`str`**, **`sta`**, **`tou`**, **`spd`**, **`lck`**, **`cou`**, and **`skl`**, make their mutator methods **`private`** and have each of them check whether its input value are in the valid range (1-20). Mutator methods whose inputs are not in the valid range must return **`false`**. Otherwise, they must return **`true`**.

Write a public method *rest()*, which resets **`pow`** to the value of max. power.

Write a public method *reducePow()* that reduce the current **`pow`** by a **`double`** value taken as its only input argument.

Furthermore, provide a public method called *setAttr()* that takes in the 7 **`int`** values to be assigned to **`str`**, **`tou`**, **`sta`**, **`spd`**, **`lck`**, **`cou`**, and **`skl`**, and makes use of all private mutator methods to set the values appropriately. This method must

return **`false`** if any one of the private mutators return **`false`**. Otherwise, it must return **`true`**.

11. Add *toString()* to Wrestler in the previous problem so that it returns a *String* in the following format: **`Wrestler: [name]`**. For example, **`System.out.print(x)`** would print **`Wrestler: Hulk Hogan`** if **`x`** refers to a *Wrestler* object with the instance variable **`name`** being "Hulk Hogan".

12. The fighting between two wrestlers in a wrestling match in our wrestling simulator is simulated in turns. Each turn follows the following steps.

    1) Determine who initiates the assault.
    Each turn starts with determining which wrestler does the attacking. For each wrestler, a random **`double`** number *d* is uniformly drawn from 0 to (the wrestler's **`spd`**)+(the wrestler's **`lck`**+the wrestler's **`cou`**)/8.0. The wrestler with the bigger *d* takes the *attack chance*. Redo if the two wrestlers have got the same values of *d*. (A tie in the value of two randomly picked **`double`** is rare.)

    2) Counterattack.
    Before the actual attack takes place, the attacker might loose his *attack chance* if the defender successfully counterattacks. For each wrestler, a random **`double`** number *c* is uniformly drawn from 0 to (the wrestler's **`skl`**+(the wrestler's **`lck`**)/8.0). A successful counterattack takes place if the defender has a bigger *c* than the attacker. When the counterattack takes place, the original defender takes the *attack chance* from the attacker.

    3) Deal the damage.
    The wrestler with the *attack chance* (*w1*) can now deal some damage to the other wrestler (*w2*). A **`double`** number *dmg* is calculated from:

    $$dmg = (\texttt{str} \text{ of } w1 + 10) - (\texttt{tou} \text{ of } w2) + e$$

    where *e* is a random **`double`** number uniformly taken from 0 to (**`cou`** of *w1*)/4.0+(**`skl`** of *w1*)/4.0-(**`skl`** of *w2*)/4.0+(**`lck`** of *w1*)/8.0-(**`lck`** of *w2*)/8.0

If *dmg* is less than 0, *dmg* is set to 0. Then, *dmg* is subtracted from *w2*'s current **pow**.

4) Account for tiredness.
After the damage dealing step, both wrestlers get some reduction in their **pow** values. The reduction *r* in **pow** is a **double** value and is calculated by:

$$r = (\text{current } \mathbf{pow}/\text{Max. Power}) \times 20 \times (\mathbf{sta}+1)^{-1}$$

5) Check whether the power is depleted.
A wrestling match is over if, after a turn is finished, **pow** values of any one of the two wrestlers are less than or equal to 0. Otherwise, a next turn is executed following similar steps, with the **pow** value of each wrestler remains what it is at the end of the current turn.

If the match is over, a wrestler with positive **pow** is the winner. If there is no winner, the match is called a tie.

Now, create a class called *FightingRule* that contains *non-static* methods needed for commencing a wrestling match according to the steps above. These methods include:

```
int determineAttacker(Wrestler w1, Wrestler w2)
```

The method returns 1 if **w1** takes the attack chance as the result of step 1). It returns 2 if **w2** takes the *attack chance*.

```
boolean canCounter(Wrestler w1, Wrestler 2)
```

The method assumes that **w1** is the attacker and **w2** is the defender. It returns **true** if a counterattack takes place as the result of step 2). Otherwise it returns **false**.

```
double dealDamage(Wrestler w1, Wrestler w2)
```

The method assumes that after step 2) in the current turn **w1** has the *attack chance*. It returns the value of inflicted damage according to the rule explained in step 3). This method also reduce **pow** of **w2** according to the damage inflicted.

```
void exhaust(Wrestler w)
```

The method reduces the value of **pow** of **w** according to the tiredness explained in step 4).

```
boolean isKO(Wrestler w)
```

The method returns true if **pow** of **w** is less than or equal to 0.

13. Create a class called *WrestlingMatch*. An object of this class represents a wrestling match between two wrestlers.

    The data members of this class are:

    ```
    private Wrestler red;
    private Wrestler blue;
    private int maxStat;
    private FightingRule rule;
    ```

    The two instance variables **red** and **blue** refer to the wrestlers involved in this match. The instance variable **maxStat** determines the maximum value that **str**, **tou**, **sta**, **spd**, **lck**, **cou**, and **skl** of each wrestler can be summed to. If a wrestler has such a sum exceeding **maxStat**, that wrestler is disqualified for the match. The instance variable **rule** is a *FightingRule* object adopted by the match.

    Write a constructor that takes an **int** value and a reference to a *FightingRule* object. It set **maxStat** to that **int** value and set **rule** to the *FightingRule* object.

    Write a method **public boolean isQualified(Wrestler w)** that returns **true** if w is qualified for the current match. Otherwise, it returns **false**.

    Write a method **public boolean setWrestlers(Wrestler w1,Wrestler w2)** that assigns the wrestler referred to by **w1** to **red** and the wrestler referred to by **w2** to **blue**. The method must return **false** if one of the wrestlers are disqualified. Otherwise, it returns **true**.

    Write a method **public Wrestler commence()** that simulates the current wrestling match based on the two objects of *Wrestler* referred to by **red** and **blue** according to the rules defined in **rule**. The method must make use of the five methods provided

in **`rule`**. Basically, the method introduces a loop inside which each turn of the match is simulated based on the methods provided in the *FightingRule* object. The loop is exited when there is a winner to the match or the match is known as a tie. It returns a reference to the *Wrestler* object who wins the match, or it returns **`null`** if the match is a tie. Inside the method *println()* should be used to echo the commentary of what has happened in the match out on screen. Design your own style of match commentary.

14. Write a program called ManualSimulation.java that performs the following steps:

    - Create an instance of *FightingRule* and have it be referred to by a variable **`currentRule`**.
    - Create an instance of *WrestlingMatch* and have it be referred to by a variable **`match`**. This match should have its **`maxStat`** at 70. Apply the instance of *FightingRule* created in the previous step. Use the constructor of *WrestlingMatch*.
    - Create the following wrestlers with the attributes specified.

Name: Tiger Mask

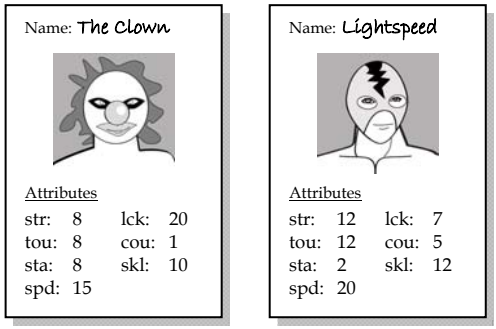
Attributes

str:	16	lck:	4
tou:	10	cou:	8
sta:	10	skl:	10
spd:	12		

Name: Nightmare

Attributes

str:	10	lck:	7
tou:	18	cou:	7
sta:	12	skl:	2
spd:	14		



- Set the wrestlers in **`match`** to "Tiger Mask" and "Nightmare". If any one of the wrestlers is disqualified due to their attributes, report it on screen. Simulate the match for the winner.
- Set the wrestlers in **`match`** to "The Clown" and "Lightspeed". If any one of the wrestlers is disqualified due to their attributes, report it on screen. Simulate the match for the winner.
- Rest the winners of the two matches by using *rest()* defined in *Wrestler*.
- Simulate a match between the winners of the two matches.

15. Explain what are needed to be done to the wrestling simulation if the formulas used in step 1) to step 4) explained in problem 12 are changed. List the names of the files whose contents need modification according to the change.

Chapter 12: Inheritance

Objectives

Readers should

- Understand the concept and role of inheritance.
- Be able to design appropriate class inheritance hierarchies.
- Be able to make use of inheritance to create new Java classes.
- Understand the mechanism involved in instance creation of a class inherited from another class.
- Understand the mechanism involved in method invocation from a class inherited from another class.

Inheritance: Creating Subclasses from Superclasses

Inheritance is an ability to derive a new class from an existing class. That new class is said to be a *subclass*, or *derived class*, of the class it is derived from, which is called *superclass*, or *base class*. A subclass can be thought of as an extension of its superclass. It inherits all attributes and behaviors from its superclass. However, more attributes and behaviors can be added to existing ones of its superclass.

Let's look at a simple example of how we create subclasses. Although it is not going to be very useful in any real programs, this should serve as the first example of class inheritance that gives you a first look at how corresponding class definition can be written. Suppose that we have a class called *C12A* whose definition is:

```
public class C12A
{
        public int x;
        public double d;
        public double f(){
            return x*d;
        }
}
```

Now, let's say that we would like to create a new class *C12B* that inherits all attributes and behaviors from *C12A* with no extra attributes or behaviors added. Writing the class definition of *C12B* does not involve repeating the code in *C12A*. We use the keyword **`extends`** to let Java know that *C12B* inherits from *C12A*. With no additional attributes or behaviors, the class definition of *C12B* can be simply written as:

```
public class C12B extends C12A
{
}
```

This very short code segment is already a valid class definition of *C12B* that we want. We could now write a program that makes use of *C12B*. Consider the following program and its output.

```
public class InheritanceDemo0                                    1
{       public static void main(String[] args)                   2
        {                                                        3
            C12B b = new C12B();                                 4
            b.x = 2;                                             5
            b.d = 1.5;                                           6
            System.out.println("b.f() = "+b.f());                7
        }                                                        8
}                                                                9
```

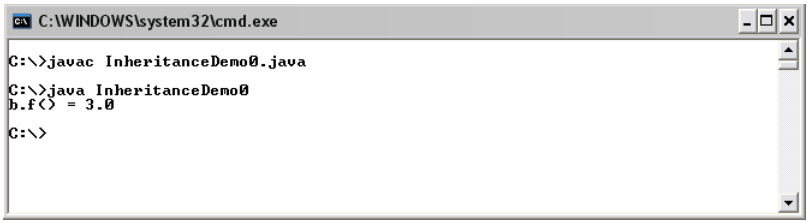


Figure 186: Creating and using a subclass without any additional data members and methods

We can see from the above program that an object of the class *C12B* contains the instance variables **`x`** and **`d`**, as well as *f()*, without having to explicitly write these attributes and behaviors in the class definition of *C12B*. *C12B* is called a subclass of *C12A*. On the other hand, *C12A* is called the superclass of *C12B*.

A class inheritance diagram can be used to show the relationship among classes. To show that *C12B* is extended or inherited from *C12A*, a diagram in the figure below can be used.

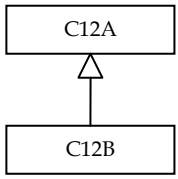


Figure 187: Diagram representing relationship between a subclass and its superclass

However, the real benefit of inheritance is not creating a new class that behaves exactly the same way as its superclass but creating a new class contains all attributes and behaviors of its superclass together with some additional attributes and behaviors. For example, if we would like to create a new class called *C12C* which has all the attributes and behaviors of *C12A* but with a name for each instance of *C12C*, we could extends *C12A* with an additional instance variable of type *String* as well as some appropriate methods. Such a class could be written as:

```
public class C12C extends C12A
{
        public String name;
        public String toString(){
            return name+":"+x+","+d;
        }
}
```

From the class definitions of the three classes: *C12A*, *C12B*, and *C12C*, their structures can be depicted as the following figure.

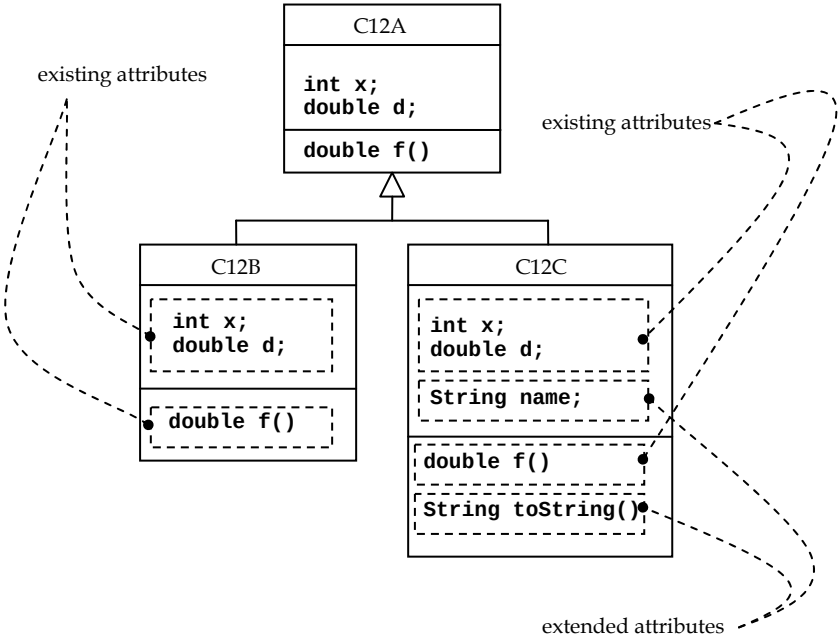


Figure 188: Class inheritance diagram relating C12A, C12B, and C12C

Now consider the following program and observe its output.

```
public class InheritanceDemo01                                          1
{                                                                       2
        public static void main(String[] args)                         3
        {                                                               4
            C12C c = new C12C();                                        5
            c.x = 5;                                                    6
            c.d = 2.0;                                                  7
            c.name = "An object with a name";                           8
            System.out.println("c.f() = "+c.f());                       9
            System.out.println("c.name = "+c.name);                     10
            System.out.println(c);                                      11
        }                                                               12
}                                                                       13
```

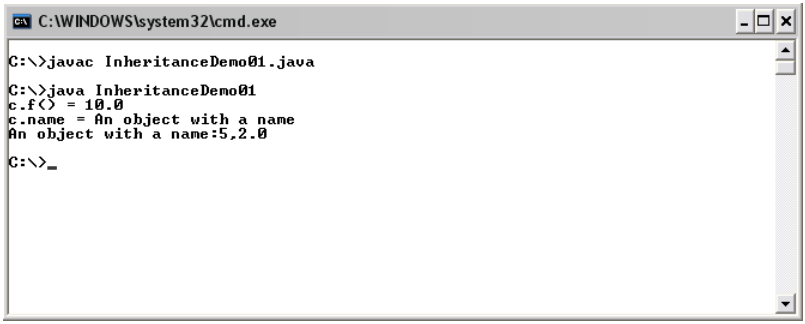


Figure 189: Subclass with additional data members and methods

On line 5, a variable **`c`** is created and is referred to a new object of *C12C*. The statements line 6 and line 7 assign values to instance variables which are defined in the class definition of *C12A*, the superclass of *C12C*. On line 8, the instance variable **`name`** which is a part extended in *C12C* from *C12A* is assigned with a *String*. On line 9, the existing method *f()* in the superclass is called, while on line 10, the instance variable **`name`** is used, and on line 11, Java automatically calls *toString()* defined in *C12C* in response to **`System.out.println(c);`**.

Note that multiple-inheritance is not allowed. A subclass can only extend from a single superclass, while a superclass can have more than one subclass inherited from the class.

Designing Class Inheritance Hierarchy

Using inheritance effectively in creating new classes does not involve only the Java syntax used in defining the new class but also organizing classes in a hierarchical manner. A good class hierarchy helps us understand the relationship among classes. Superclasses are always more general than subclasses since a subclass possesses everything that its superclass has while it can also possess additional attributes and behaviors. There is an *is-a (is-an) relationship* between a subclass and its superclass. That means an object of a subclass "*is an*" object of its superclass.

A subclass can be inherited further by other classes. This makes a hierarchy of classes. The figure below shows an example of a hierarchy among some quadrilaterals.

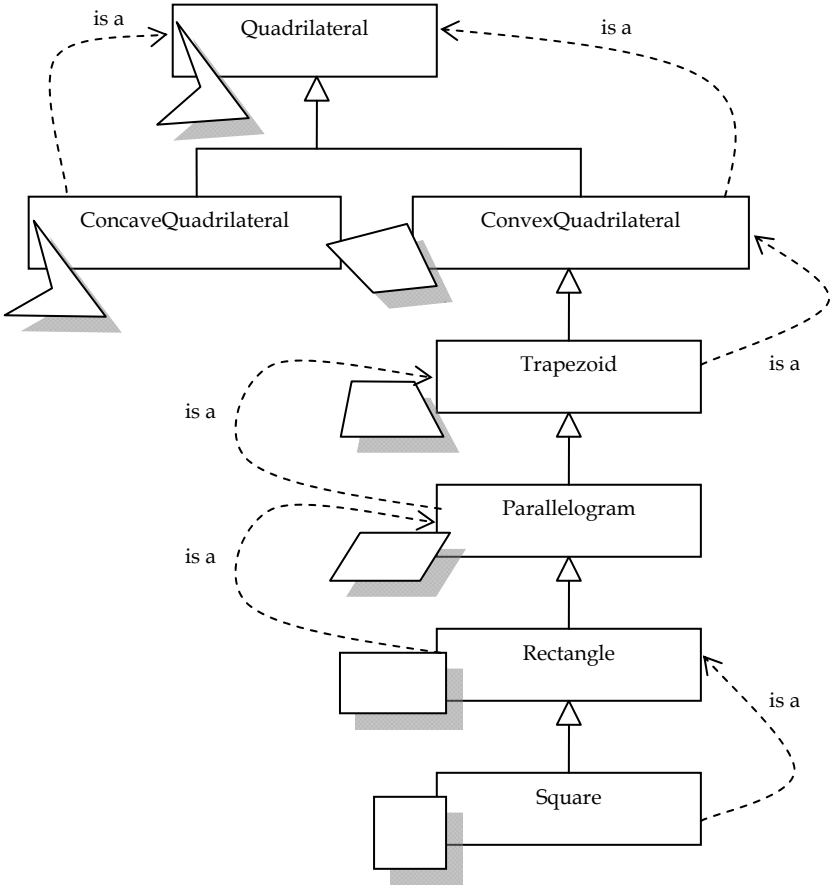


Figure 190: Class inheritance hierarchy of quadrilateral

Here are the definitions of the shape shown above.

- Quadrilateral: A polygon with four sides and four vertices.

- Convex quadrilateral: A quadrilateral whose every internal angle is at most 180 degrees and every line segment between two vertices of the quadrilateral remains inside or on the boundary of the quadrilateral.
- Concave quadrilateral: A quadrilateral that is not convex.
- Trapezoid: A convex quadrilateral in which one pair of opposite sides is parallel.
- Parallelogram: A trapezoid whose opposite sides have equal length, opposite angles are equal and the diagonals bisect each other.
- Rectangle: A parallelogram where each angle is a right angle.
- Square: A rectangle where four sides have equal length.

A good class hierarchy must conform to that each subclass instance 'is a' superclass instance. From the class inheritance hierarchy in Figure 190, we can say that a square is a rectangle, a rectangle is a parallelogram, and so on. According to the hierarchy, a square is also a quadrilateral, or we can say that a square is a more specific case of a quadrilateral. A quadrilateral is the most general case of all the shapes in the hierarchy. Therefore, every shape listed below the quadrilateral class is a quadrilateral with additional attributes and behaviors.

Example 97: Designing classes for a map and a maze

Suppose we are part of a team to develop a computer game in each stage of which a rectangular map for that stage is to be displayed on screen. It is designed that a map is composed of $N \times M$ tiles where N is the width of the map of that stage and M is the height. When a stage is played, each tile of the map of that stage will be rendered with a picture represented with an *Image* object (an instance of the *Image* class in the java.awt package). A map for a stage could look like what is shown in Figure 191.

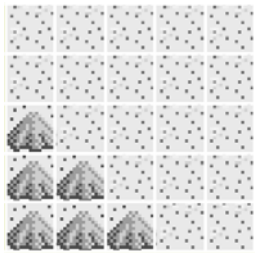


Figure 191: A map with 5 × 5 tiles

If we create a class called *GameMap* to represent the map and another class called *GameTile* to represent the tile according to what mentioned above, the relationship of *GameMap* and *GameTile* together with Image could be designed as shown in Figure 192.

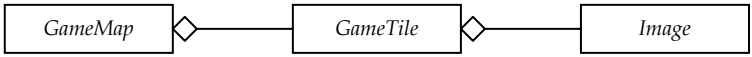


Figure 192: Relationships among GameMap, GameTile and Image

Now, let's extend our design so that our classes can represent a maze which can be considered as a special case of a map but with at least two extra characteristics which are:

1. A maze has a starting tile where players are placed at the tile when a stage is entered or restarted as well as a destination tile where the stage is cleared when players arrive at.

2. A maze is composed of special kind of tiles instead of tiles used in a general map. When residing in each of these special tiles, a player can only move to adjacent tiles only in the directions that are not blocked.

A maze could look like the one shown in Figure 193. In the figure, the picture on the left shows the rendered result of a maze and the one on the right indicates which directions of each tile are blocked (shown with solid black line along the border of each block).

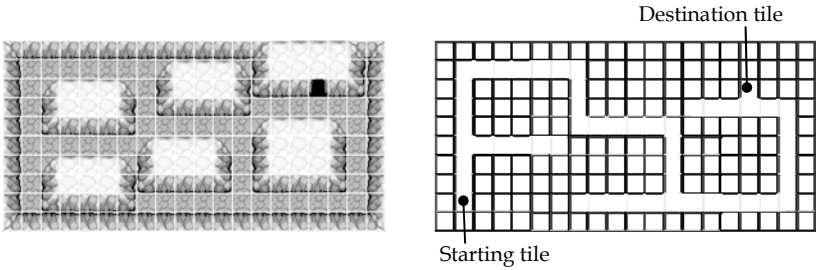


Figure 193: A maze with 20 × 10 tiles

There are two classes that we should create in addition to *GameMap* and *GameTile*. Let's name the class representing a maze *GameMaze* and the special kind of tile *RestrictedGameTile*. Since a *GameMaze* object is a *GameMap* object with additional data members as well as possible additional methods supporting the members, it makes sense that we create the class by extending from the class *GameMap*. With the same reasoning, the *RestrictedGameTile* should be extended from the regular *GameTile* with additional data members containing information about which directions are blocked together with some possible additional related methods.

Therefore, the relationships among the classes should be like the ones in Figure 194.

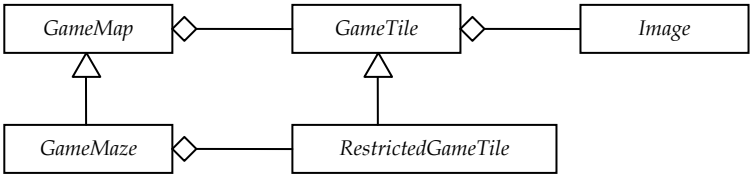


Figure 194: Relationships among GameMap, GameTile, Image, GameMaze, and RestrictedGameTile

---

Access Levels and Subclasses

As discussed in Chapter 11, when data members and methods are declared or defined in a class, creator of the class can use access level modifiers to limit the access made to them. Now that we know about subclasses, let's re-visit the meaning of the access levels again.

Data members and methods with the **`public`** modifier can be accessed using the dot operator from anywhere. For data members and methods with the **`private`** modifier, they can be accessed only from inside the same class where they are declared. In the case of the **`protected`** modifier, data members and methods with this modifier can be accessed from

within the class where they are declared as well as from within any subclasses of the class where they are declared. If the access level modifiers are omitted, the access level used is called the default level, which is rather similar to the **`private`** access level but the access is also extended to any classes in the same package.

Access Level	Accessing Class		
	current class	subclass	other
public	☑	☑	☑
protected	☑	☑	☒
default	☑	☒	☒
private	☑	☒	☒

Table 12: Summary of access levels (packages are ignored)

Table 12 summarizes the four access level without paying attention to the concept of Java package which will not be covered in this book. A table cell marked with ☑ means that the corresponding accessing class can access resources with the corresponding access level. In other words, the dot operator can be used in the code listed inside the accessing class to access resources with the corresponding access level directly. A table cell marked with ☑ means that we cannot use the dot operator inside the accessing class in order to access resources with that access level directly.

Keyword 'super'

Data members or methods in a superclass having either **`public`** or **`protected`** access levels can be accessed directly from inside the class definitions of its subclasses by referring to the name of the data members or methods to be accessed. It is possible that there is a variable declared within a subclass that has the same name as a variable declared in its superclass. In this case, the **`super`** keyword is used with the dot operator to indicate the variable declared in the superclass, while referring to the variable with only its name indicate the variable declared in the subclass. For example, let's suppose that there is a variable called **`var`** declared inside both the definition of the subclass and the definition of the superclass. From within the definition of the subclass, the expression

super.var will refer to the one in the superclass, while **var** will refer to the one in the subclass. This works very similarly to the case of the **this** keyword.

Example 98: Using super

Consider the following classes and observe the output of *SuperDemo*.

```
public class MySuperclass                                              1
{                                                                      2
       public int a = 1;                                               3
       public int b = 2;                                               4
       public void f(){                                                5
         System.out.println("\tf() of MySuperclass is called.");       6
       }                                                               7
       public void g(){                                                8
         System.out.println("\tg() of MySuperclass is called.");       9
       }                                                               10
}                                                                      11
```

```
public class MySubclass extends MySuperclass                           1
{                                                                      2
       public int a = 9;                                               3
       public void f(){                                                4
           System.out.println("\tf() of MySubclass is called.");       5
       }                                                               6
       public void test(){                                             7
           System.out.println("a = "+a);                               8
           System.out.println("b = "+b);                               9
           System.out.println("super.a = "+super.a);                   10
           System.out.println("super.b = "+super.b);                   11
           System.out.println("f()");                                 12
           f();                                                        13
           System.out.println("g()");                                 14
           g();                                                        15
           System.out.println("super.f()");                           16
           super.f();                                                  17
           System.out.println("super.g()");                           18
           super.g();                                                  19
       }                                                               20
}                                                                      21
```

```
public class SuperDemo                                                 1
{                                                                      2
       public static void main(String[] args)                         3
       {                                                               4
           MySubclass y = new MySubclass();                            5
```

(continued on next page)

(continued from previous page)

```
            y.test();                                    6
        }                                                7
}                                                        8
```

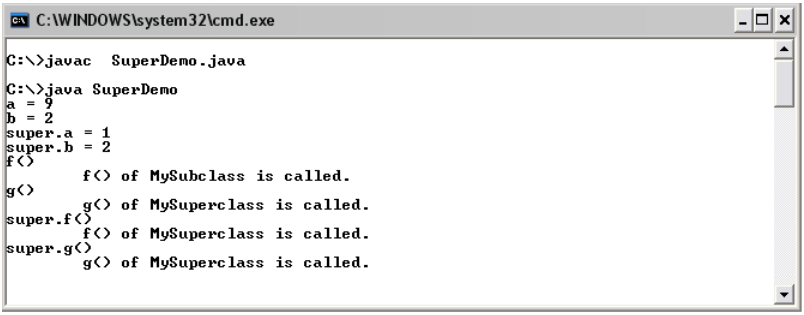


Figure 195: Using the keyword super to identify variables of the superclass

Let's start by looking at the *main()* method of *SuperDemo*. The statement on line 5 instantiates a new object if the class *MySubclass*. Next, it invoke *test()* from that object. Considering the definition of *Mysubclass*, the *test()* method prints the values of **`a`**, **`b`**, **`super.a`** and **`super.b`** on line 8 to line 11. We can observe from the output that the value of **`a`** is **`9`**, while **`super.a`** is **`1`** since the former one is the variable **`a`** inside the subclass and the later one is the variable **`a`** from the superclass. The values of **`b`** and **`super.b`** are both **`2`**. Both expressions refer to the same variable **`b`** declared in the superclass since there is no ambiguity here.

On line 13, line 15, line 17, and line 19, **`f()`**, **`g()`**, **`super.f()`**, and **`super.g()`** are called respectively. We can see that *f()* in the subclass is called as the result of **`f()`**, while *f()* in the superclass is called as the result of `super.f()`. Both **`g()`** and **`super.g()`** invoke *g()* in the superclass since there is no method with that name defined in the subclass.

Polymorphism

A variable whose type is a class can refer to an object of that class as well as an object of any one of its subclasses (no matter how many levels down the inheritance hierarchy). However, the opposite is not true. A variable of a class cannot be used to refer to an object of its superclass, just like when a variable cannot be used to refer an object whose class is different from the variable type.

In OOP, this ability of an object of a class can be appeared as and used like an object of its subclass is called *polymorphism*.

Recall the class hierarchy of *C12A*, *C12B*, and *C12C* shown in Figure 188 and consider the following code segments.

```
C12A a = new C12A();
C12B b = new C12B();
C12C c = new C12C();
a = b;
a = c;
```

This code segments will not cause any compilation errors even though the data types between **a** and **b** and between **a** and **c** are different. The reason is because the data type of **a** which is *C12A* is the superclass of both *C12B* and *C12C*.

```
C12A a1 = new C12B();
C12A a2 = new C12C();
```

The initialization of **a1** and **a2** shown above is also correct.

```
C12A [] a = new C12A[3];
a[0] = new C12A();
a[1] = new C12B();
a[3] = new C12C();
```

These statements also work fine since each element in the array can refer to an object of *C12A* as well as an object of its subclass.

However, the following program cannot be compiled successfully.

```
C12A a = new C12A();
C12B b = new C12B();
C12C c = new C12C();
b = a;
c = a;
b = c;
```

There are three problems among these statements. First, `b = a` cannot be done since the object referred to by `a` is a subclass of the data type of `b`. Second, `c = a` cannot be done for a similar reason. Lastly, `b = c` cannot be done either since the data type of the objects referred to be `b` and `c` are not directly related at all. They only share the same superclass.

Method Overriding

When a method that has already been defined in a class is redefined in its subclass using the same identifier and the same list of input arguments, it is called that the method defined in the subclass *overrides* the one defined in its superclass. When the method with that identifier is invoked from an object of a class, Java looks for the method definition in that class first and then looks inside the superclass later.

In the case that the data type of a variable is not the same as the data type of the object that it is currently referring to (i.e. the object belongs to a class that directly or indirectly inherits from the class that is the type of the variable), Java chooses the method definition based on the type of the object, not the type of the variable.

Example 99: Overriding methods

Observe the class definition of *C12D*.

```
public class C12D extends C12A
{
        public double y;
        public double f(){
            return x*d+y;
        }
}
```

Then, consider the following program and its output.

```
public class MethodOverridingDemo1                                    1
{                                                                     2
       public static void main(String[] args)                        3
       {                                                              4
           C12A a = new C12A();                                       5
           a.x = 2;                                                   6
           a.d = 1.0;                                                 7
                                                                      8
           L12D d = new C12D();                                       9
           d.x = 2;                                                   10
           d.d = 1.0;                                                 11
           d.y = 2.5;                                                 12
                                                                      13
           System.out.println("a.f()="+a.f());                        14
           System.out.println("d.f()="+d.f());                        15
                                                                      16
           a = d;                                                     17
           System.out.println("a.f()="+a.f());                        18
       }                                                              19
}                                                                     20
```

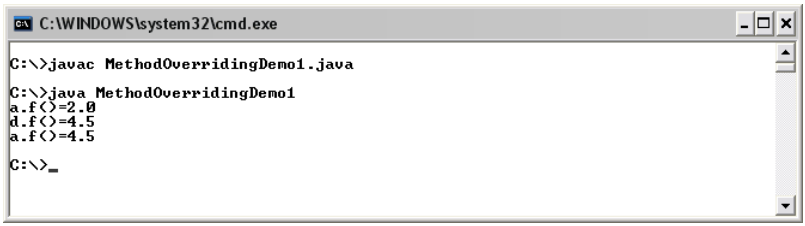


Figure 196: Demonstration of method overriding

The important points in this example is the methods invoked when **`a.f()`** and **`d.f()`** are called. On line 5, **`a`** is made to refer to an C12A object. On line 9, **`d`** is made to refer to an C12D object. Therefore, **`a.f()`** on line 14 invokes *f()* from C12A, and **`d.f()`** on line 15 invokes *f()* from C12D. However, after that, **`a`** is made to refer to an *C12D* object (the same object that **`d`** refers to), which is a valid operation since *C12D* is a subclass of *C12A*. Consequently, **`a.f()`** on line 18 invokes *f()* from *C12D* instead of *f()* from *C12A*.

In conclusion, you have to keep in mind that it is not the variable type that determines the method invocation, but the type of object to which the variable refers.

Polymorphism also enables the ability of objects belonging to different types to respond to method calls of methods with the same identifier, each one according to an appropriate type-specific behavior. Java does not have to know the exact type of the object in advance, so this behavior can be implemented at run time. This is called *late binding* or *dynamic binding*.

Example 100: A book shop

Consider the following class definitions.

```
public class BookItem
{
        protected String name;
        protected double listedPrice;

        public BookItem(String name,double price){
            this.name = name;
            listedPrice = price;
        }
        public double getSellingPrice(){
            return listedPrice;
        }
        public double getListedPrice(){
            return listedPrice;
        }
        public String toString(){
            return "BookItem:"+name;
        }
}
```

```
public class UsedBook extends BookItem
{
        protected double discountFactor;

        public UsedBook(String name,double price,double
discountFactor){
            super(name,price);
            this.discountFactor = discountFactor;
        }
        public double getSellingPrice(){
```

(continued on next page)

(continued from previous page)

```
            return (1-discountFactor)*listedPrice;
        }
        public String toString(){
            return "UsedBook:"+name;
        }
}
```

```
public class RareBook extends BookItem
{
        protected double premiumFactor;

        public RareBook(String name,double price,double
premiumFactor){
            super(name,price);
            this.premiumFactor = premiumFactor;
        }
        public double getSellingPrice(){
            return (1+premiumFactor)*listedPrice;
        }
        public String toString(){
            return "RareBook:"+name;
        }
}
```

The three class definitions listed above show that the class *BookItem* is the superclass of the other two classes, *UsedBook* and *RareBook*. Each of the three class has its own implementation of *getSellingPrice()* and *toString()*. As mentioned, which implementation of these two methods is invoked depends on the type of the object from which the method is called. The following program uses the three classes we have just defined.

```
import java.io.*;                                                    1
public class BookShop                                                2
{                                                                    3
  public static void main(String[] args) throws IOException          4
  {                                                                  5
    BufferedReader stdin =                                           6
      new BufferedReader(new InputStreamReader(System.in));          7
    System.out.print("Number of books:");                           8
    int nBooks = Integer.parseInt(stdin.readLine());                 9
    BookItem [] bookInventory = new BookItem[nBooks];                10
    int i=1;                                                         11
    while(i<=nBooks){                                                12
      bookInventory[i-1] = getBook(i);                               13
      i++;                                                           14
    }                                                                15
```

(continued on next page)

(continued from previous page)

```
    showBookInfo(bookInventory);                                    16
                                                                    17
  }                                                                 18
  public static BookItem getBook(int i) throws IOException{         19
      BufferedReader stdin =                                        20
        new BufferedReader(new InputStreamReader(System.in));       21
      BookItem b;                                                   22
      while(true){                                                  23
        System.out.println("---------------\nBook #"+i);           24
        System.out.print("Type(1=Regular,2=Used,3=Rare):");        25
        int type = Integer.parseInt(stdin.readLine());              26
        if(type<1 || type>3){                                       27
          System.out.println("Invalid type.");                      28
          continue;                                                 29
        }                                                           30
        System.out.print("Book name:");                             31
        String name = stdin.readLine();                             32
        System.out.print("Listed price:");                          33
        double price = Double.parseDouble(stdin.readLine());        34
        double factor;                                              35
        switch(type){                                               36
          case 1:                                                   37
            b = new BookItem(name,price);                           38
            break;                                                  39
          case 2:                                                   40
            System.out.print("Discount factor:");                   41
            factor = Double.parseDouble(stdin.readLine());          42
            b = new UsedBook(name,price,factor);                    43
            break;                                                  44
          case 3:                                                   45
            System.out.print("Premium factor:");                    46
            factor = Double.parseDouble(stdin.readLine());          47
            b = new RareBook(name,price,factor);                    48
            break;                                                  49
          default:                                                  50
            b = null;                                               51
        }                                                           52
        break;                                                      53
      }                                                             54
      return b;                                                     55
  }                                                                 56
  public static void showBookInfo(BookItem [] bookInventory){       57
    System.out.println("#####################################");    58
    for(int i=0;i<bookInventory.length;i++){                        59
      System.out.println("Item #"+i+":\t"+bookInventory[i]);        60
      System.out.print("Listed Price:\t");                         61
      System.out.println(bookInventory[i].getListedPrice());       62
      System.out.print("Selling Price:\t");                        63
      System.out.println(bookInventory[i].getSellingPrice());      64
```

(continued on next page)

(continued from previous page)

```
      System.out.println();                                          65
    }                                                                66
    System.out.println("####################################");     67
  }                                                                  68
}                                                                    69
```

Let's look at *main()* first. The program asks the user to input the number of books to be stored in an array of *BookItem*. Then, the user is asked to input the information of each book. Finally, the program prints out information associated with each element in the array. We can see that at the compilation of the code, the program has no way to know the type of each element in the array **`bookInventory`** except that it has to be of the type *BookItem* or one of its subclasses.

On line 12 to line 15, a *while* loop is used to gather information of each array element. Inside the method *getBook()*, an object of type either *BookItem*, *UsedBook*, or *RareBook* is created and returned from the method. The returned object is referred to as an element of **`bookInventory`**. Notice that the return type of *getBook()* is *BookItem*, with which it is also perfectly correct for the method to return an object of its subclass.

Polymorphism is used in *showBookInfo()* since the program does not know in advance about the exact type of each object in **`bookInventory`**. For each element in the array, *toString()* is called inexplicitly on line 60, *getListedPrice()* is called on line 62, and *getSellingPrice()* is called on line 64. Late binding through polymorphism in Java makes the program know at run-time which implementation of *toString()* and *getSellingPrice()* (i.e. the implementations in the superclass or the subclass) to be invoked based on the type of the object from which the methods are called. For *getListedPrice()*, it is only implemented in the superclass. Therefore, there is no confusion on which method to be invoked.

Observe an output of BookShop.java shown below. Pay attention to the value of the selling price for each element printed on screen.

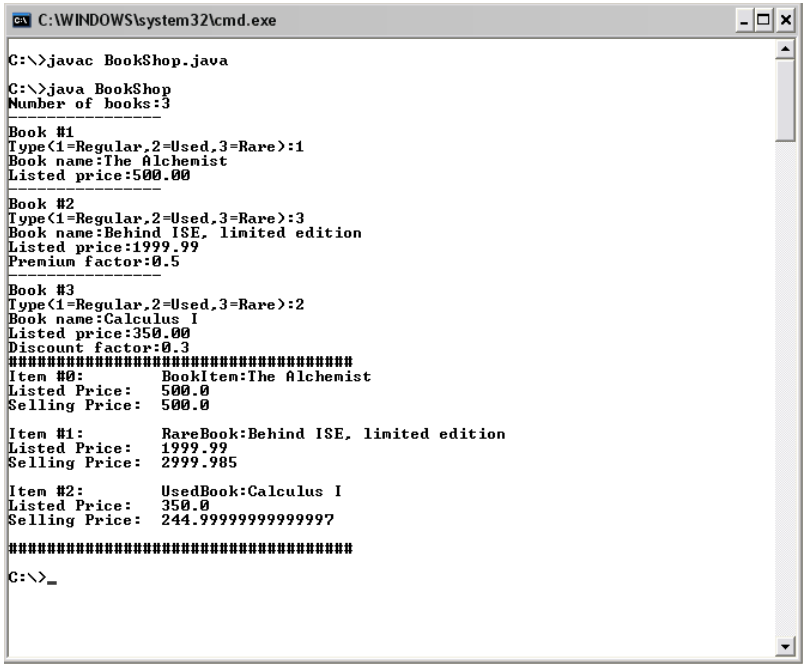


Figure 197: A program that makes use of late-binding

Note that we will not cover *abstract classes* and *interface* in this book. These concepts are usually used with polymorphism in order to obtain more benefits from late binding.

Instance Creation Mechanism

When an instance or object of a class is created, if there is no explicit call to any constructors of its superclass, the no-argument constructor of the superclass is called automatically before the execution of any statements in the subclass's constructor (if there are any).

To demonstrate this mechanism, let's look at the following example.

Example 101: Creation of inherited instances

```
public class C12E
{
        public C12E(){
            System.out.println("\tC12E() is called.");
        }
}
```

```
public class C12F extends C12E {}
```

```
public class C12G extends C12E
{
        public C12G(){
            System.out.println("\tC12G is called.");
        }
        public C12G(String s){
            System.out.println("\tC12G(String s) is called.");
        }
        public C12G(int i){
            super();
            System.out.println("\tC12G(int i) is called.");
        }
}
```

C12F and *C12G* are subclasses of *C12E*. Let's consider the following program. Pay attention to messages printed on screen when each object is created.

```
public class CreationDemo1                                          1
{                                                                   2
        public static void main(String[] args)                     3
        {                                                           4
            System.out.println("1)-----------------");              5
            C12F f = new C12F();                                    6
            System.out.println("2)-----------------");              7
            C12G g1 = new C12G();                                   8
            System.out.println("3)-----------------");              9
            C12G g2 = new C12G("Hello");                            10
            System.out.println("4)-----------------");              11
            C12G g3 = new C12G(8);                                  12
        }                                                           13
}                                                                   14
```

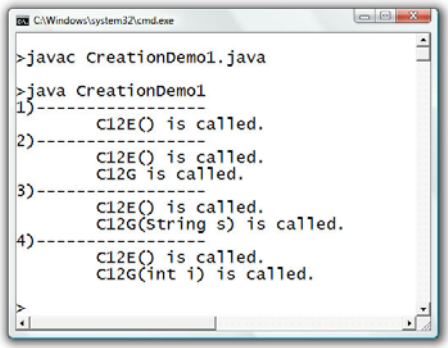


Figure 198: Constructors invocation during instantiation of objects

When an object of *C12F* is created on line 5, since there is no implementation of any constructor in *C12F*, the no-argument constructor of its superclass, *C12E()*, is invoked automatically.

When an object of *C12G* is created, if either *C12G()* or *C12G(String s)* is called, the no-argument constructor of its superclass, *C12E()*, is again invoked automatically since there is no explicit call to the constructor of *C12E*. If *C12G(int i)* is called, the constructor of *C12E* is not called automatically since it is explicitly called by the first statement of *C12G(int i)*.

In order to explicitly call any constructor of the superclass, the *super()* statement can be used but it has to be used as the first statement in the constructor. Otherwise, it will result in a compilation error. The following class cannot be compiled successfully.

```
public class C12H extends C12E
{
    public C12H(){
            System.out.println("\tC12H is called.");
            super(); // This causes a compilation error
    }
}
```

Exercise

1. Why do we need an ability to create new classes from other base classes?
2. Is it possible to create a new class called *SmartString* that inherits from the *String* class?
3. Consider the following class definitions.

```
public class Ex12_3
{
    public int x;
    public int y;
}
```

```
public class Ex12_3sub
                extends Ex12_3
{
  public int y = 1;
  public int z = 1;
}
```

What is the output of the following program?

```
public class Ex12_3run
{
    public static void main(String[] args)
    {
        Ex12_3 a = new Ex12_3();
        Ex12_3sub b = new Ex12_3sub();
        System.out.println(a.x+","+a.y);
        System.out.println(b.x+","+b.y+","+b.z);
    }
}
```

4. Consider the following class definition.

```
public class Ex12_4
{
    public String s;
    public Ex12_4(String s){
        this.s = s;
    }
}
```

As well as,

```
public class Ex12_4a extends Ex12_4
{
	public String s;
	public Ex12_4a(String s){
		super(s);
		this.s = revert(s);
	}
	public String revert(String s){
		if(s.length()<=1) return s;
		return s.charAt(s.length()-1)+
			revert(s.substring(0,s.length()-1));
	}
	public void printS(){
		System.out.println(s);
		System.out.println(super.s);
	}
}
```

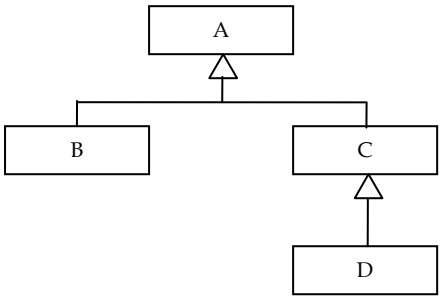
What is the output of the following program?

```
public class Ex12_4run
{
	public static void main(String[] args)
	{	Ex12_4a a = new Ex12_4a("What?");
		System.out.println(a.s);
		a.printS();
	}
}
```

5. Draw a class inheritance diagram describing the relationship among the following items.

   Webpage, Mouse, Monitor, Word Processor, Router, Input Device, Operating System, Ink-jet Printer, USB Mouse, Computer System Component, Output Device, Web Browser, Hardware, Flat-panel Display, Joystick, CRT Monitor, Laser Printer, Software, Network Peripheral, Data, Application, System Software, Printer

6. The following class inheritance diagram defines the relationship among *A*, *B*, *C*, and *D*.



Which ones of the following statements can be compiled successfully?

a. Statement: `A a1 = new B();`
b. Statement: `B b1 = new C();`
c. Statement: `B b2 = new D();`
d. Statement: `C c1 = new D();`
e. Statement: `C c2 = new A();`
f. Statement: `A a2 = new D();`

7. How many (immediate) base classes can a class have?

What are the Next Steps?

What you have learned in this textbook should get you started in creating computer programs using Java programming language in order to solve some problems in the form of Java console applications. What are more important than the syntaxes on Java are the basic concepts required for writing computer programs in other high-level languages as well as scripting some special-purpose application software packages.

Still, if ones would want to further their study of computer programming in order to develop more complex software applications according to modern programming methodologies, here are some useful topics with suggested readings given in the trailing parentheses.

- Additional object-oriented programming concepts including abstraction and interfaces ([Wei2008]).
- Error and exception handling framework (Coh2004)
- Software testing ([Mcc2009])
- Framework-based programming such as Graphical User Interface (GUI) framework, Collection framework, and Input/Output framework ([Jia2002])
- Design patterns ([Fre2004], [Gam1994])
- Program development in other platforms including web-based programming ([Seb2010], [Bei2008], [Nix2009], [Wel2008]), programming for embedded systems ([Sim1999], [Bar1999]), programming for mobile devices and smart phones ([All2010],[Bur2010]).

References

[All2010] Allan, A., Learning iPhone Programming: From Xcode to App Store, O'Reilly Media, 2010

[Bar1999] Barr, M., Programming Embedded Systems in C and C ++, O'Reilly Media, 1999

[Bei2008] Beighley, L., and Morrison, M., Head First PHP & MySQL, O'Reilly, 2008

[Bur2005] Burnette, E., Eclipse IDE Pocket Guide, O'Reilly Media, 2005

[Bur2010] Burnette, E., Hello, Android: Introducing Google's Mobile Development Platform, Pragmatic Bookshelf, 2010

[Coh2004] Cohoon, J., and Davisdon, J., Java Program Design, McGraw-Hill, 2004

[Dan2004] D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., and McCarthy, P., The Java Developer's Guide to Eclipse, Addison-Wesley Professional, 2004

[Fow2003] Fowler, M., UML Distilled: A Brief Guide to the Standard Object Modeling Language, Addison-Wesley Professional, 2003

[Fre2004] Freeman, E., Freeman, E., Bates, B., and Sierra, K. , Head First Design Patterns, O'Reilly, 2004

[Gam1994] Gamma, E., Helm R., Johnson, R., and Vlissides M., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional, 1994

[Jia2002] Jia, X., Object Oriented Software Development Using Java , Addison Wesley, 2002

[Mcc2009] McCaffrey, J., Software Testing: Fundamental Principles and Essential Knowledge, BookSurge Publishing, 2009

[Mil2006] Miles, R., and Hamilton, K., Learning UML 2.0, O'Reilly Media, 2006

[Mya2008] Myatt, A., Pro Netbeans IDE 6 Rich Client Platform Edition, Apress, 2008

[Nag2008] Nagal, C., Evjen, B., Glynn, J., Skinner, M., and Watson, K., Wrox Professional C# 2008, Wiley Publishing, 2008

[Nix2009] Nixon, R., Learning PHP, MySQL, and JavaScript: A Step-By-Step Guide to Creating Dynamic Websites, O'Reilly Media, 2009

[Pil2005] Pilone, T., and Pitman, N., UML 2.0 in a Nutshell, O'Reilly Media, 2005

[Pre2010] Pressman, R., Software Engineering: A Practitioner's Approach, McGraw-Hill, 2010

[Ros2007] Rosen, K., Discrete Mathematics and its Applications, McGraw-Hill, 2007

[Seb2009] Sebesta, R., Concepts of Programming Languages, Addison Wesley, 2009

[Seb2010] Sebesta, R., Programming the World Wide Web, Addison Wesley, 2010

[Sim1999] Simon, D., An Embedded Software Primer, Addison-Wesley Professional, 1999

[Wei2008] Weisfeld, M., The Object-Oriented Thought Process, Addison-Wesley Professional, 2008

[Wel2008] Welling, L., and Thomson, L. , PHP and MySQL Web Development, Addison-Wesley Professional, 2008

Index