UMBC
Training Centers
6996 Columbia Gateway Drive
Suite 100
Columbia, MD 21046
Tel: 443-692-6600
http://www.umbctraining.com

JAVA PROGRAMMING

Course # TCPRG2000
Rev. 3/8/2016

This Page Intentionally Left Blank

Course Objectives

- At the conclusion of this course, students will be able to:
  - Compile and run a Java application.
  - Understand the role of the Java Virtual Machine in achieving platform independence.
  - Navigate through the API docs.
  - Use the Object Oriented paradigm in Java programs.
  - Understand the division of classes into Java packages.
  - Use Exceptions to handle run time errors.
  - Select the proper I/O class among those provided by the JDK.
  - Use threads in order to create more efficient Java programs.

This Page Intentionally Left Blank

Table of Contents

CHAPTER 1: INTRODUCTION

CHAPTER 2: LANGUAGE COMPONENTS

CHAPTER 3: OBJECT-ORIENTED PROGRAMMING

CHAPTER 4: METHODS

CHAPTER 5: ARRAYS

CHAPTER 6: ENCAPSULATION

CHAPTER 7: INHERITANCE & POLYMORPHISM

CHAPTER 8: ABSTRACT CLASSES AND INTERFACES

CHAPTER 9: EXCEPTIONS

CHAPTER 10: INPUT AND OUTPUT IN JAVA

CHAPTER 11: COLLECTIONS

CHAPTER 12: NETWORKING

CHAPTER 13: THREADS

This Page Intentionally Left Blank

Chapter 1: Introduction

What is Java?

- Java is a Programming Language developed at Sun Microsystems beginning in 1991.
  - The credit for Java is usually given to James Gosling who led a group of engineers on a project named OAK. The project, based on C++, had as its goal the control of consumer devices.
  - The first public release of Java was in 1996 and was called the **J**ava **D**evelopment **K**it (**JDK**). It has seen become known as the **S**oftware **D**evelopment **K**it (**SDK**).

- Java has been described in various ways.
  - **Simple** - Java is simple compared to C++.
  - **Object-Oriented** - A style of programming emphasizing the marriage of data and methods rather than algorithms directly.
  - **Distributed** - Java has built-in networking capabilities.
  - **Interpreted** - The Java interpreter executes bytecodes on any machine to which the interpreter has been ported.
  - **Robust** - During bytecode generation, Java checks for any possible errors rather than allowing an error to propagate to the run-time environment.
  - **Architecture Neutral** - When a Java program is compiled, the result is an architecture independent bytecode file. The bytecode file is designed to run on what is called the Java Virtual Machine (JVM).
  - **Secure** - Java promises that a program cannot overwrite memory outside of its own process space, or read or write local files when invoked through an applet in a Web Browser.
  - **Portable** - Many of the problems from C and C++ are removed. For example, an `int` is always 32 bits, regardless of the machine. Strings are encoded using Unicode. Floating-point numbers are stored in a fixed format.

Versioning

- SUN delivers different versions of the SDK in terms of their functionality. The versions are listed below.
  - The **Micro Edition (Java ME)**
    - Specifically addresses the vast consumer space.
    - This covers the range of extremely tiny commodities such as smart cards or pagers, all the way up to the set-top box, an appliance almost as powerful as a computer.
  - The **Standard Edition (Java SE)**
    - Features a development and deployment environment designed from the ground up for the Web.
    - It provides cross-platform compatibility, safe network delivery, and smart card to supercomputer scalability.
  - The **Enterprise Edition (Java EE)**
    - Defines the standard for developing and deploying enterprise applications.
    - It includes the latest versions of Enterprise JavaBeans, JavaServer Pages, Java Servlet APIs, the Java API for XML Parsing (JAXP), the Java Authentication and Authorization Service (JAAS) API, and other API's.

The Java Virtual Machine

- The **J**ava **V**irtual **M**achine (**JVM**) is a computer (in software) with Java bytecodes as the instruction set.
  - A developer uses an editor to create a Java source file whose name ends with the `.java` extension.
  - A Java Compiler converts this to a file of bytecodes whose name ends with the `.class` extension.
- The JVM interprets and verifies the bytecodes of a file as it is loaded.
  - If the class file passes the verifier, it is loaded and translated into specific OS instructions and executed on the target machine.
- The compiled `.class` files are architecture neutral.
  - The language that produces the bytecode is irrelevant.
  - Currently, any compiler can output Java bytecode.
    - Therefore, the target platform for Java code is not any particular machine.
    - It is a virtual machine (i.e., any computer that can translate the bytecodes into native code).
- Performance at runtime can be somewhat slower than traditional compilation because of verification, conversion to native machine code, array bounds checking and automatic garbage collection.

Writing a Java Program

- Writing a Java program requires some tools and administration.

- The process generally proceeds as follows.
  - Enter Java text into a file with any text editor.
    - Java source files must end with the suffix `.java`.
    - The name of the file must match the name of the public class.

Hello.java

```
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

  - Compile the source code.
    - The Java compiler is named `javac`.

```
javac Hello.java
```

    - If there are no errors, the output from the compiler is a file whose name has the name of the `.java` file but with the `.class` extension. (`Hello.class` in this case.)
    - This file contains platform independent bytecodes.
  - Execute the program using the Java interpreter on the `.class` file.
    - The name of the interpreter is `java`.

```
java Hello
Hello World
```

Writing a Java Program

- The `Hello` program is intentionally simple in order to pick out the basic parts of a Java program.

- All Java files consist of one or more classes.
  - Of these, exactly one of them can be made public.
  - The name of the file must be the name of the public class.

- When a Java source file is compiled, a `.class` file is built for each class in the source file.

- A class can have more than one method.

- When an application is executed, the starting point for the application is always the following `main` method.

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

  - `void`: means that the method, `main`, does not return a value. However, some methods do return a value.
  - `public`: is a keyword that controls access to this code. Code in other classes has access to a public method. There are other access levels other than public that will be introduced later in the course.
  - `static`: implies the method is not associated with an object but instead is associated with a class.
  - The `println` method is a way of sending output to the standard output file.

Packages

- A **package** is a collection of related classes that:
  - makes the classes easier to find and use;
  - assists in avoiding naming conflicts; and
  - assists in controlling access to the classes.

- To specify the package of a class, you put a `package` statement at the top of the source file of the class.
  - If a package statement is not used, the class ends up in the **default package**, which is a package that has no name.
    - The previous `Hello` application is an example of a class in the default package.
    - It is recommended that all classes generally belong in a named package.

- Classes in a package need to be placed in a directory structure that matches the package name.
  - For example, many of the classes in this course could have been grouped into a package named `examples`.
  - Since the course has many chapters, it might be better to subdivide the classes into sub-packages.
    - The classes in this chapter can be further grouped into a package named `examples.intro`.
  - Therefore, if we were to place `Hello.java` into this package we would use the following statement at the top of the class.

```
package examples.intro;
```

Packages

- If the `Hello` class is specified as being in the `examples.intro` package, the compiled `Hello.class` needs to be placed in a directory named `intro`, which in turn needs to be placed in a directory named `examples.`
  - Although the source file, `Hello.java,` can be stored anywhere, it is recommended that source files be maintained in the same package structure as the `.class` files.

- Below is a new version of `Hello.java` that has been placed in the `examples.intro` package.

Hello.java

```
package examples.intro;
public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello Again");
    }
}
```

  - `Hello.java` is located in the following sub-directory of the labfiles directory:
    - `examples\intro`
  - Therefore, to compile `Hello.java,` we would change to the above directory and run the compiler, resulting in a file named `Hello.class` in the `examples\intro` directory.
    - `cd javalabs\examples\intro`
    - `javac Hello.java`

Packages

- Keep in mind that we now have two `Hello.class` files.
  - `C:\javalabs\Hello.class`
  - `C:\javalabs\examples\intro\Hello.class`

- To properly reference a class file in Java, you must supply the fully qualified name of the class, which always includes the package name.
  - *`the.package.name.ClassName`*

- To demonstrate how each of the above `.class` files can be run, we will first change the command prompt to the `C:\javalabs` directory.
  - Since the first `Hello` class we defined had no package specified, it is in the default package that has no name.
    - Therefore, to run the first Hello program we would simply type the following at the prompt.

      ```
      C:\javalabs>java Hello
      ```

    - This results in the following output.

      ```
      Hello World
      ```

  - The second `Hello` class we defined specified a package of `examples.intro.`
    - So, to run the second `Hello` program we would type

      ```
      C:\javalabs>java examples.intro.Hello
      ```

    - This results in the following output.

      ```
      Hello Again
      ```

Packages

- Both the JVM and the Java compiler rely on an environmental variable named `CLASSPATH` to locate class files.
  - If the `CLASSPATH` is not specified, it defaults to the current directory.
  - To simplify our development environment, we will set the `CLASSPATH` to the directory containing the `examples` directory.
    - There is a file in the `javalabs` directory named `setenv.cmd,` which should have the following entry added.

      ```
      set CLASSPATH=C:\javalabs
      ```

    - Running this script will set up our environment so that the JVM and the compiler will always begin looking for `.class` files it needs in the `javalabs` directory.
    - Keep in mind that for the remainder of the course, all classes will be organized into packages, and that to run a program you will need to provide the fully qualified name of the class on the command line.

      ```
      java examples.intro.Hello
      ```

- The environment being set by the `setenv.cmd` script is only set for that shell window.
  - If a new window is opened, the `sentenv.cmd` script would need to be run inside of that window as well.

Simple Java Programs

- Below is a second program that builds upon the previous information by providing additional statements.

AddIntegers.java

```
package examples.intro;
public class AddIntegers {
    public static void main(String args[]) {
        int x = 10;
        int y = 20;
        System.out.print("Sum of ");
        System.out.print(x);
        System.out.print(" and ");
        System.out.println(y);
        System.out.print("is ");
        System.out.println(x + y);
    }
}
```

  - Compile the class by typing the following on the command line.

```
javac AddIntegers.java
```

  - Run the program by typing the following on the command line.

```
java examples.intro.AddIntegers
```

  - The output from executing the program is shown below:

```
Sum of 10 and 20
is 30
```

  - Note that the `println` method adds a newline character to the output, whereas the `print` method does not.

- The above example defined a few variables of type `int`.

  - Soon you will see additional data types.

Simple Java Programs

- The program on the previous page could have been written in many different ways.

AddAgain.java

```
package examples.intro;
public class AddAgain {
    public static void main(String args[]) {
        int x = 10, y = 20;
        System.out.print(x + " + " + y);
        System.out.println(" is " + ( x + y));
    }
}
```

- The output of the above program is shown below.

```
10 + 20 is 30
```

- When a `String` data type is added to any other data type, the + operator concatenates the values rather than adding them as shown in the example below.

Concatenation.java

```
package examples.intro;
public class Concatenation {
    public static void main(String args[]) {
        int x = 10, y = 20;
        System.out.println(x + y + " is " +  x + y);
    }
}
```

- The output of the above program is: `30 is 1020`
  - Developers need to be aware of the dual purpose of the + operator in Java.

Simple Java Programs

- A method in Java cannot stand alone.
  - It must be embedded in some class.
  - The following code will not compile because the `main` method is not embedded in a class.

```
public static void main(String args[]) {
    int x = 10, y = 20;
    System.out.println(x + y + " is " +  x + y);
}
```

- The program below has a `main` method but the parameter list is incorrect.
  - This program will compile but cannot be executed as an application.

Wrong.java

```
package examples.intro;
public class Wrong {
    public static void main() {
        int x = 10, y = x * 2;
        System.out.println(x + y + " is " +  x + y);
    }
}
```

Exercises

1. The following program contains multiple errors.

   - Correct each of the errors until the program compiles and executes without any errors.

     - A copy of this file can be found in the `starters` directory for this chapter.

```
package starters.intro.ex1;
Public Class MyNewClass {
    public void static main(String s){
        integer a = 5
        system.out.println("a = ", a);
    }
}
```

2. Write a program that includes in its `main` method, the three lines of code shown below.

```
int a = 17, b = 4, c;
c = a + b;
System.out.println(a + " + " + b + " = " + c);
```

   - Run the program and interpret the results.

   - Reusing the variable c, add similar statements for the `/` and `%` operators so that the results of all three calculations appear in the output.

Chapter 2:
Language Components

Primitive Data Types

- Java supports a wide range of data types. The eight fundamental (or primitive) data types are shown below.

```
byte    myByte    = 0;             //  8 bits
short   myShort   = 15000;         // 16
int     myInteger = 42;            // 32
int     myHexInt  = 0xA;           // 32
long    myLong    = 8000000000L;   // 64
float   myFloat   = 3.14159f;      // 32
double  myDouble  = 2.3E24;        // 64
boolean myTruth   = true;          //  1
char    myChar    = 'A';           // 16
char    yourChar  = '\t';          // 16
char    aChar     = '\u03C0';      // 16
```

- Constants of any of the above types can be created by using the keyword `final`.

```
final double TAX_RATE = 0.06;
```

- Single quotes are used for a literal `char`.

```
char letter = 'A';
```

- Local data (variables declared inside methods) must be initialized before being used.
  - Object data, which we will discuss later, will be automatically initialized.

- The `print` and `println` methods can have any of the above types as an argument.
  - This is known as **method overloading** and will be discussed in more detail in a later chapter.

Primitive Data Types

- Characters use the **Unicode** character set so that non-English characters can be easily encoded
  - Unicode is a standard designed to consistently encode characters used in written languages throughout the world.
  - The Unicode standard uses hexadecimal to express a character.
  - When the specification for the Java language was created, the Unicode standard was accepted and the `char` primitive was defined as a 16-bit data type, with characters in the hexadecimal range from 0x0000 to 0xFFFF.
    - A `char` became insufficient to define all characters in use throughout the world as the Unicode standard was extended to over one million characters.
    - The definition of a character in the Java programming language could not be changed from 16 bits to 32 bits without causing millions of Java applications to no longer run properly.
    - To correct the definition the characters with values that are outside of the 16-bit range, and within the range from 0x10000 to 0x10FFFF, are called supplementary characters and are defined as a pair of char values.
- More information about Unicode and its implementation within the Java programming language can be found at the following URLs.
  - *http://www.unicode.org*
  - *https://docs.oracle.com/javase/tutorial/i18n/text/unicode.html*

Primitive Data Types

- The program below illustrates some of the primitive Java data types.

DataTypes.java

```
package examples.language;
public class DataTypes {
    public static void main(String args[]) {
      int a = 10, b = 3;
      System.out.print(a + " * " + b + " = ");
      System.out.println(a * b);

      double x = 3.5;
      System.out.print(a + " + " + a + " * " + x);
      System.out.println(" = " + (a + a * x));

      System.out.println("A can be represented as:");
      System.out.print("A" + " or " + 'A' + " or ");
      System.out.println('\u0041');

      System.out.println("Water Freezes @ 32\u00B0");
      boolean c;
      c = a == b;
      System.out.print("The statement " + a);
      System.out.println(" == " + b + " is " + c);

      c = a != b;
      System.out.print("The statement " + a);
      System.out.println(" != " + b + " is " + c);
    }
}
```

- Below is the output from the above program.

```
10 * 3 = 30
10 + 10 * 3.5 = 45.0
A can be represented as:
A or A or A
Water Freezes @ 32°
The statement 10 == 3 is false
The statement 10 != 3 is true
```

Comments

- Java allows three distinct comment types.
  - C style
    - Typically be used to comment a large section of text.

```
/*   a large
     section
     of text
*/
```

  - C++ style
    - Used most effectively to comment a single line or the last portion of a line.

```
//  This is an entire line of commentary
//  So is this

int x = 0;
x = x + 1;  // add one to x
```

  - Javadoc style
    - Is used to define special comments, which can be used by the javadoc utility in the SDK to produce HTML documentation for your Java code (i.e., to produce Java documentation).

```
/**
 * This describes the variable
 * declaration below
 */
int x = 10;
```

Control Flow Statements

- The statements inside of a method are generally executed in the order that they appear.
  - Control flow statements, however, break up the flow of execution.
- This section describes the following three types of control flow statements supported by the Java programming language:
  - Decision-making statements (if-then, if-then-else, switch),
  - Looping statements (for, while, do-while), and
  - Branching statements (break, continue).
- All of the control flow constructs contain opening and closing braces that are optional, provided that the body of the construct contains only one statement.
  - If the construct contains more than one statement in its body, then the opening and closing curly braces are required.
  - Deciding when to omit the braces is a matter of personal preference.
    - Omitting them can make for a common mistake if a second statement is later added and the now required braces are forgotten.
    - This omission will generally mean you'll just get the wrong results.

The `if` Statement

- The `if` statement tells your program to execute a certain section of code only if a particular test evaluates to `true`.
  - The following example tests to see if the value of x is even.

    ```
    int x = 10;
    if ( x % 2 == 0 )
        System.out.println(x + " is even");
    ```

    - If this test evaluates to `false` (meaning `x` is not even), control jumps to the end of the `if` statement.
  - The `if-else` statement provides an alternate path of execution when an `if` statement evaluates to `false`.

    ```
    int x = 7;
    if ( x % 2 == 0 )
        System.out.println(x + " is even");
    else
        System.out.println(x + " is odd");
    ```

- The following program prints out a message based on what time of day it is.

`IfTest.java`

```
package examples.language;
public class IfTest {
    public static void main(String args[]) {
        int hour = 2;
        if (hour < 6)
            System.out.println("Too early");
        else if (hour < 12) {
            System.out.print("Good morning, ");
            System.out.println("how are you?");
        }
        else if (hour < 18)
            System.out.println("Good afternoon");
        else
            System.out.println("Good evening");
    }
}
```

The `switch` Statement

- A `switch` statement can be used as an alternative to a set of `if else` constructs.
  - It is often a matter of style with regard to which construct to use.

`SwitchTest.java`

```
package examples.language;
public class SwitchTest {
    public static void main(String args[]) {
        int i = 5;
        System.out.print(i + " is: ");
        switch(i) {
            case 0:
                System.out.print("very ");
            case 1:
            case 2:
                System.out.println("small");
                break;
            case 3:
            case 4:
            case 5:
                System.out.println("bigger");
                break;
            case 6:
            case 7:
                System.out.println("large");
                break;
            default:
                System.out.println("biggest");
        } // end of switch
    }
}
```

- The primitives used in an expression in a `switch` statement must be a single `char`, `byte`, `short`, or `int`.
  - The expression may also be a `String` object.

The `while` and `do while` Statements

- A `while` loop continually executes a block of statements while a particular condition is true.

WhileTest.java

```
package examples.language;
public class WhileTest {
    public static void main(String args[]) {
        int low = 10, high = 20, sum = 0;
        int save = low;
        while( low <= high ) {
            sum += low++;
            System.out.println("Partial Sum:" + sum);
        }
        System.out.print("Sum of ints from ");
        System.out.print(save + " to " + high);
        System.out.println(" is " + sum);
    }
}
```

- There is also a `do while` construct that differs from both the `while` in that the loop test is performed at the bottom of the loop as shown in the example below.

DoWhileTest.java

```
package examples.language;
public class DoWhileTest {
    public static void main(String args[]) {
        int i = 1, sum = 0;
        do {
            sum += i++;
        } while( i <= 10 );
        System.out.println(sum);
    }
}
```

The `for` Statement

- A `for` statement provides a compact way to iterate over a range of values.
  - The general form of the `for` statement is as follows:.

    ```
    for ( initialization; test; modification ) {
        // body of the loop goes here;
    }
    ```

    - The *`initialization`* expression initializes the loop; it's executed once, as the loop begins.
    - When the *`test`* expression evaluates to `false`, the loop terminates.
    - The *`modification`* expression is invoked after each iteration of the loop; this expression may increment or decrement a value.
- The example below uses a `for` loop that sums the integers from 1 to 50.

Sums.java

```
package examples.language;
public class Sums {
    public static void main(String args[]) {
        int sum = 0;
        for (int i = 1; i <= 50; i = i + 1) {
            sum = sum + i;
        }
        System.out.println("Sum = " + sum);
    }
}
```

  - Notice the declaration of the variable `i` in the initialization part of the `for` loop.
    - Note that the scope of that variable is within the loop itself.

The break Statement

- In Java, any looping construct can have the flow altered through the use of a break or continue statement.
  - break breaks the current loop.
  - continue continues with the next iteration of the loop.

- For example, the code below breaks out of the for loop when the value of i squared is greater than 500.

BreakTest.java

```
package examples.language;
public class BreakTest {
    public static void main(String args[]) {
        int i;
        for ( i = 1; i <= 100; i = i + 1){
            if ( i * i > 500 )
                break;
        }
        System.out.println("i is "  + i);
    }
}
```

- In the event that you need to break from inside of a nested loop, you can break with a label.

NestedBreakTest.java

```
package examples.language;
public class NestedBreakTest {
    public static void main(String args[]) {
        int i,j;
outer:  for (i = 0; i < 10; i++)
           for ( j = 0; j < 10; j++)
              if ( i + j > 15 )
                 break outer;
        System.out.println("i is "  + i);
    }
}
```

The `continue` Statement

- To demonstrate the `continue` statement, the code below shows one way of adding up all those numbers below `100` that are not divisible by `3`. It uses the `%` operator that gives the remainder of dividing one number by another.
  - When a `continue` is executed in a `for` loop, the next executed statement is the modification part of the `for` loop.
  - If a `continue` is executed in a `while`, the next executed statement is the `while` test.

ContinueDemo.java

```
package examples.language;
public class ContinueDemo {
    public static void main(String args[]) {
        // for loop example
        int i, sum = 0;
        for ( i = 1; i <= 100; i = i + 1){
            if ((i % 3 ) == 0 )
                continue;
            sum = sum + i;
        }
        System.out.println("sum "  + sum);

        //while loop example
        i = 0;
        sum = 0;
        while (++i <= 100){
            if ((i % 3 ) == 0 )
                continue;
            sum = sum + i;
        }
        System.out.println("sum "  + sum);
    }
}
```

Operators

- Operators are special symbols that perform specific operations on one, two, or three operands, and then return a result.

1	[ ] . ()	array subscript class method qualifier function invocation
2	! ~ ++ -- + - (cast) new	logical not 1's complement auto increment auto decrement unary plus unary minus explicit conversion object creation
3	* / %	multiplication division modulus
4	+ -	addition subtraction
5	<< >> >>>	left shift right shift sign fill right shift 0 fill
6	< <= > >= instanceof	less than less than or equal greater than greater than or equal run time type id
7	== !=	equality inequality
8	&	Bitwise and
9	^	Bitwise exclusive or
10	|	Bitwise inclusive or
11	&&	Logical and
12	||	Logical or
13	?:	Ternary conditional operator
14	= += -= *= /= %= &= |= ^= <<= >>= >>>=	assignment operator other assignment ops

- All operators are left to right associative except those on lines 2 and 14.

Casts and Conversions

- Both variables and constants have a type in Java.
  - Since Java is very strict about type conversions, there are a few rules that you will need to understand.

- Each of the following initializations is illegal because each violates the size restriction of the data type.

```
char  c = 65536;
byte  b = 128;
short s = 32768;
int   i = 2147483648;
```

- All integral arithmetic is carried out as `int` unless one of the operands is a `long`, in which case, the result is a `long`. This means that all of the assignments below will cause a compiler error.

```
byte b = 5;
b = b + 10;          // compiler error

short s = 5;
s = s + 5;           // compiler error

long el = 10;
int val = 20;
val = val + el;      // compiler error
```

- There are times when you need to perform some of the operations above.
  - For these instances, you can use a **cast**, an explicit instruction to the compiler to make a conversion.

Casts and Conversions

- It is often necessary to perform an explicit cast when performing integral arithmetic as shown below.

```
byte b = 5;
b = (byte) (b + 10);      // compiles ok

short s = 5;
s = (short) (s + 5);      // compiles ok

long el = 10;
int val = 20;
val = (int) (val + el);  // compiles ok
```

- Interestingly, the following will not cause a compiler error but will result in an uncaught overflow error in your program.

```
byte b = 127;
b +=1
System.out.println(b); // -128
```

- Since constants have a type, the first line below will cause a compiler error.

```
float x = 2.0;   // compiler error since 2.0 is double:

float x = 2.0f;        // one fix
float x = (float) 2.0; // another fix
```

- Whenever any double precision computations are carried out, the compiler follows this rule.
  - If either of the operands is a `double`, the other is converted to `double`; otherwise,
  - If either of the operands is a `float`, the other is converted to a `float`.

Keywords

- The table below is a list the Java keywords that are reserved. This means you cannot use them as names in your Java programs. In addition, `true`, `false`, and `null` are reserved words and, therefore, you cannot use them as names in your programs either.

Java Keywords			
abstract	double	int	super
assert	else	interface	switch
boolean	enum	long	synchronized
break	extends	native	this
byte	final	new	throw
case	finally	package	throws
catch	float	private	transient
char	for	protected	try
class	goto *	public	void
const *	if	return	volatile
continue	implements	short	while
default	import	static	
do	instanceof	strictfp **	

* indicates a keyword that is not currently used
** indicates a keyword that was added for Java 2

Exercises

1. Write a Java program which uses a `for` loop to compute the sum of the odd integers from `1` to `100`.

2. Use nested `for` loops to produce the following output.

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
6 5 4 3 2 1
7 6 5 4 3 2 1
```

3. Use nested `while` loops to produce the following output.

```
1 2 3 4 5 6 7
1 2 3 4 5 6
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

4. Print a table showing the even integers between `20` and `60` in the first column, their squares in the 2nd column, and their cubes in the 3rd column.

Exercises

5. Add statements inside the `for` loop below such that the following output is produced.

```
for (int i = -4; i <= 4; i++) {
}

-4 is negative and even
-3 is negative and odd
-2 is negative and even
-1 is negative and odd
0 is even
1 is positive and odd
2 is positive and even
3 is positive and odd
4 is positive and even
```

6. Write a program that uses a `while` loop to compute `10` factorial.

   - What is the largest factorial that can be fit inside a variable of type `int`?

7. Write a program that produces those sets of consecutive integers totaling exactly `10,000`.

8. Given the following variable declarations:

```
short x = 10;
byte b = 20;
float f = 2.0f;
long val = 1;
```

   - What are the types of the following expressions?

```
x + x
x + b
x + f
10 + 'A'
x + b + val
```

This Page Intentionally Left Blank

Chapter 3: Object-Oriented Programming

Defining New Data Types

- Primitive types are provided in Java for the solution of general-purpose problems.

- For solutions to specific problems, Java allows you to define your own data types. New data types are created using the `class` keyword.

- New data types are created in order to fill in the gaps left by general purpose programming languages.

- Once a new data type has been defined, programmers can create instances of them. Each instance of a new data type is called an **object**.

- In the real world, an object is an idea or a "thing" that is physical or conceptual. Software is easier to understand and maintain when code can be mapped to the physical problem for which the software is designed.

  - For example if you are simulating an elevator in software, it would be nice to have an object of data type `Elevator` with operations `up`, `down`, `pickup`, `discharge`, and `stop`.

- All objects have two general characteristics: **behavior** and **attributes**.

  - For example, any object of type `Automobile` would have the following common attributes and behavior.

    - attributes: `weight, length, color, fuelCapacity`
    - behaviors: `start, stop, accelerate, turn`

Defining New Data Types

- Suppose we model a Person.
  - Each object of type `Person` would have the following characteristics.
    - attributes: `age, weight, height, eyeColor`
    - behavior: `eat, sleep, talk, walk`

- An object's attributes and behavior are described by a `class` definition.
  - Sometimes attributes are called **fields** or **data**.
    - The collection of all the data for an object is called the **state** of the object.
  - A behavioral characteristic is called a function, an action, or a **method**.
    - The complete set of public methods for a class is typically called the **public interface**.

- A class can also represent an idea or a concept. Note the methods and data fields for each item below.
  - An object of type `File` would have:
    - attributes: `size, type, createDate`
    - behaviors: `open, close, read, write`
  - An object of type `Fraction` would have:
    - attributes: `numerator, denominator`
    - behaviors: `add, print, multiply`

Defining New Data Types

- In a Java Program, to define any of these new types using object-oriented principles, we would first map each of them into a `class`.
  - The `class` keyword is used to define new data types.
    - Before an `Automobile` object can be defined, a `class` named `Automobile` would be created.

      ```
      public class Automobile{
          // Attributes represented as variables here
          // Behaviors represented as methods here
      }
      ```

    - Before a `File` object can be defined, a `class` named `File` would be created.

      ```
      public class File{
          // Attributes represented as variables here
          // Behaviors represented as methods here
      }
      ```

    - Before a `Fraction` object can be defined, a `class` named `Fraction` would be created.

      ```
      public class Fraction{
          // Attributes represented as variables here
          // Behaviors represented as methods here
      }
      ```

  - Therefore, a class combines data and methods and acts as a template or blue print for constructing objects of that data type.
    - To define a new data type, you must specify both the attributes for the new data type and the methods for the new data type.
    - The packaging of these two characteristics is known as **encapsulation**.

Defining New Data Types

- Suppose you wish to define a new data type called a `Loan.`
  - The first step would be to define a class named `Loan`, as shown below.

```
// characteristics of a Loan will be placed within
// the class definition
public class Loan {

}
```

  - An object of type `Loan` would have a name, amount, interest rate, and length of loan to which it is associated. These pieces of data would be defined as variables inside of the `class` as shown below.

```
public class Loan {
    // Loan attributes declared here
    String name;
    double amount, rate;
    int years;
}
```

  - An object of type `Loan` would also have some behavior, such as the ability to set the name of the loan. Therefore, also appearing in the template for a loan would be the methods that realize the behavior of a loan.

```
public class Loan {
    // Loan attributes declared here
    String name;
    double amount, rate;
    int years;

    // Loan behavior declared here
    public void setName(String n) {
        // method functionality would be placed here
    }
}
```

Defining New Data Types

- A complete definition for the `Loan` class is shown below.

`Loan.java`

```
package examples.ooprogramming;
public class Loan {
    String name;
    double amount, rate;
    int years;

    public void setName(String n) {
        name = n;
    }
    public void setAmount(double a) {
        amount = a;
    }
    public void setRate(double r) {
        rate = r;
    }
    public void setYears(int y) {
        years = y;
    }
    public String getName() {
        return name;
    }
    public double getAmount() {
        return amount;
    }
    public double getRate() {
        return rate;
    }
    public int getYears() {
        return years;
    }
}
```

- The above class encapsulates both the data and methods of the class.
    - Note that the above class does not contain a `main` method.
    - The idea behind this class is that it can be used by any number of programs.

Defining New Data Types

- Although many programs could use the `Loan` data type, we present a small test program below. The test program simply:
  - creates a new object of type `Loan` using the `new` operator;
  - sets the data inside the new object by calling methods on the object; and
  - gets the data from the object by calling methods on the object.

LoanTest.java

```
package examples.ooprogramming;
public class LoanTest {
    public static void main(String args[]) {
        Loan myLoan = new Loan();

        myLoan.setName("James");
        myLoan.setAmount(250000);
        myLoan.setRate(4.0);
        myLoan.setYears(30);

        String theName = myLoan.getName();
        System.out.println(theName);
        // Could have been combined as
        // System.out.println(myloan.getName());

        System.out.println(myLoan.getAmount());
        System.out.println(myLoan.getRate());
        System.out.println(myLoan.getYears());
    }
}
```

- Our final topic on defining a class will be about constructors.
  - There are many additional topics to discuss about Object Orientation but, because the topic is extensive, we defer much of this information until later chapters.

Constructors

- You probably noticed that the data for a `Loan` object was given through a set of methods.

```
Loan myLoan = new Loan();
myLoan.setName("James");
myLoan.setAmount(250000);
myLoan.setRate(4.0);
myLoan.setYears(30);
```

- It might have been easier if the data could have been given during the construction of the new object.

```
Loan myLoan= new Loan("James", 250000, 4.0, 30);
```

  - Such a method is in fact called a **constructor**.
    - Classes usually provide one or more of them.
    - Full details about constructors will be given later.
  - For now we merely state that a constructor is a method that:
    - must have the same name as the class; and
    - cannot have a return value.

- Therefore, a `Loan` constructor might look like what is shown below.

```
public Loan(String n, double a, double r, int y) {
    name = n;
    amount = a;
    rate = r;
    years = y;
}
```

  - The above constructor would be added to the existing `Loan` class.

The `String` Class

- At this time, we will present some simple classes that are part of the SDK.

- The first class we will discuss is the `String` class.
  - This class has already been built. Therefore, a variety of methods already exists for this class.
  - We will demonstrate a few of them in the example that follows.
  - Keep in mind that the `String` class can be thought of as part of a library.
    - Somewhere in the SDK is a file named `String.java` that has been compiled into a file called `String.class`.
    - These files are analogous to the `Loan.java` and `Loan.class` files that we developed on the previous pages.

The `String` Class

`StringTest.java`

```
package examples.ooprogramming;
public class StringTest {
    public static void main(String args[]) {

        String name = new String("Jeremy Walker");
        System.out.println("Name is " + name);

        int len = name.length();
        System.out.println("length is " + len);

        int place = name.indexOf(' ');
        System.out.print("a space was found ");
        System.out.println("at position " + place);

        String first = name.substring(0, place);
        System.out.println("First Name is " + first);

        String last = name.substring(place + 1);
        System.out.println("Last Name is  " + last);

        char firstNameInit = first.charAt(0);
        char lastNameInit  = last.charAt(0);

        System.out.println("Initials are " +
            firstNameInit + lastNameInit);
    }
}
```

- In the above code, we have used several String methods.

```
name.indexOf(' ');
name.length();
first.charAt(0);
name.substring(0, place);
name.substring(place + 1);
```

- Note that the last two methods have the same name but different parameter lists. This is called **method overloading**.

- Note also the use of a `String` constructor.

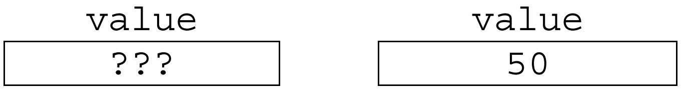
```
String name = new String("Jeremy Walker");
```

The `String` Class

- `String` objects are not primitive data. They are reference data. We will take a careful look at this.

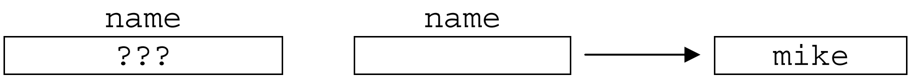
  - When one defines an `int` and then assigns a value to the storage for that `int`, the situation looks like what is shown below.

```
int value;          value = 50;
```



  - The situation is slightly different for `String` (or any other) objects. `String` objects are references to values – not values themselves.

```
String name;       name = new String("mike");
```



- String objects are immutable. This means that the actual object to which the reference points cannot be altered.

  - Therefore, in the code shown below:

```
String name = new String("Michael");
name = name + "Rodriguez";
```

    - the first line creates a new `String` object; and.
    - the second line results in `name` now referencing a completely different `String` object that is a result of concatenating `"Rodriguez"` to the value of the existing object.

- Soon, we will introduce the `StringBuffer` class, which is a better choice when a `String` needs manipulating.

String Literals

- In order to create an object in Java, you must use the new operator to explicitly create an object.

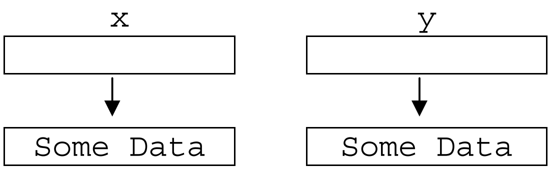
```
Loan m = new Loan();
String x = new String("Michael Saltzman");
```

- However, since String objects are so commonly used, Java relaxes this rule through the use of a literal String.

```
String x = "Hello";
String text = "This literal String is "
      + "concatenated to this literal String";
```

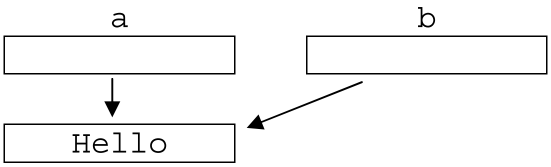
  - Keep in mind that the String class is the only class that does not require the use of the new keyword to create an object.
    - When you create a String with the new keyword, Java always creates a new object.

```
String x = new String("Some Data");
String y = new String("Some Data");
```



    - However, when you use the short form, you do not always get a new object.

```
String a = "Hello";
String b = "Hello";
```



    - In the diagram above, both a and b reference the same String object.

`String` Literals

- The use of `String` literals leads to some tricky idioms that must be understood. All of them can be demonstrated with the following program.

`StringComparisons.java`

```
package examples.ooprogramming;
public class StringComparisons {
    public static void main(String args[]){
        String x = new String("mike");
        String y = new String("mike");
        System.out.println(x == y);
        String b = "mike";
        String c = "mike";
        System.out.println(b == c);
        System.out.println(x == c);
        x = "mike";
        y = "mike";
        System.out.println(x == y);
        System.out.println(x == c);
    }
}
```

  - The output of the above program is shown below.

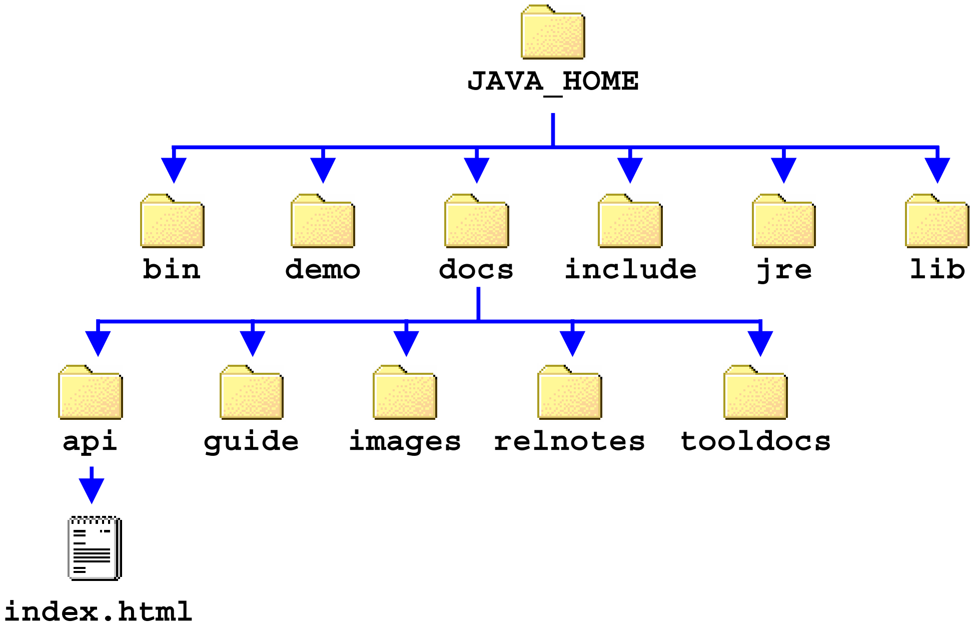
```
false
true
false
true
true
```

- To determine if the sequence of characters in two `String` objects are equal rather than testing the references, the `equals` method of the `String` class can be used rather than the `==` operator.

```
String x = new String("some data to test");
String y = new String("some data to test");
String z = new String("some Data to test");
boolean result = x.equals(y);  //would result in true
result = x.equals(z);          //would result in false
```

Documentation

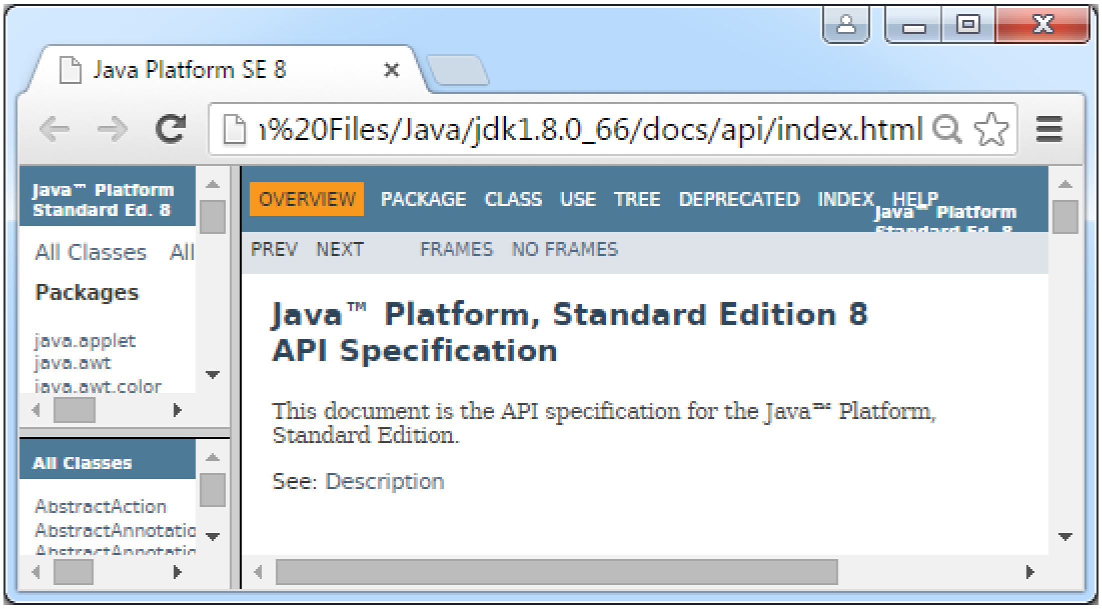
- Now that we have presented a few methods from the `String` class, how do we determine the names of all of the methods available?
  - When you download the SDK, you will also want to download the documentation files.
  - The documentation files ("docs") are delivered as a zip file. They are typically unzipped to the same directory as the SDK.
  - The picture below illustrates a partial directory structure of the SDK installation including the documentation - the "docs" directory.



  - JAVA_HOME is usually the value of an environment variable specifying the actual location and name of the directory where the SDK was installed. For example, on my machine this is `C:\Program Files\Java\jdk1.8.0_66`
    - Opening the `index.html` file (the one located in the `api` directory) in a browser should display the window shown on the next page.

Documentation

- Below is a picture of the `index.html` file.


Java Platform SE 8
ı%20Files/Java/jdk1.8.0_66/docs/api/index.html
Java™ Platform Standard Ed. 8
All Classes All
Packages
java.applet
java.awt
java.awt.color
All Classes
AbstractAction
AbstractAnnotatio
OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java™ Platform Standard Ed. 8
PREV NEXT FRAMES NO FRAMES

Java™ Platform, Standard Edition 8
API Specification

This document is the API specification for the Java™ Platform, Standard Edition.

See: Description

- Frame 1 (upper left)
  - Since the available classes number in the hundreds, they are organized into **packages** of similar classes. Each package is listed in this frame.
- Frame 2 (lower left)
  - This contains a list of all the classes provided in the SDK.
  - Clicking on a specific package name in frame 1 limits the list of classes in frame 2 to those within the specified package.
- Frame 3 (right)
  - This provides details for each class chosen from frame 2.

- As you go through this course, you will become very familiar with some of the standard Java packages.

Packages

- The first package we will study is the `java.lang` package.
  - If you wish to see only the classes in this package, just click on it in the upper left-hand frame.
  - The lower left-hand frame will list only the classes in the `java.lang` package.
  - This package is the only package that the commands, `javac` and `java,` will find without any help from the programmer.
    - For other packages, you will have to import them as you will soon see.
- The `java.lang` package consists of some fundamental classes, such as the `String` class, as well as some others that we will soon see.
- As you go through this course, you will become familiar with the core Java packages.

`java.lang`	Standard Language classes
`java.util`	Utility classes
`java.io`	Input and Output classes
`java.net`	Networking classes

The `StringBuffer` Class

- The `StringBuffer` class is another class in the `java.lang` package.

- A `StringBuffer` is like a `String`, except that it can be modified - unlike a `String`, which is immutable.
  - The principal operations on a `StringBuffer` are the `append` and `insert` methods.
    - The `append` methods always add to the end of the buffer.
    - The `insert` methods add to a specified point in the buffer.
  - The `toString` method returns the contents of the `StringBuffer` as a `String`.
  - A `StringBuffer` has both a `length` and a `capacity` method.
    - The `length` method returns the current number of characters stored in the `StringBuffer`.
    - The `capacity` method returns the number of characters that the `StringBuffer` is capable of storing before it has to dynamically resize itself to fit more.

- The example on the next page demonstrates several of the methods in the `StringBuffer` class discussed above.

The `StringBuffer` Class

- The example below demonstrates several of the methods available in the `StringBuffer` class.

`StringBufferTest.java`

```
package examples.ooprogramming;
public class StringBufferTest{
    public static void main(String args[]){
        StringBuffer sb = new StringBuffer();
        System.out.print("length:" + sb.length());
        System.out.println(" cap:" + sb.capacity());
        System.out.println();
        sb.append(123456789);
        System.out.println(sb.toString());
        System.out.print("length:" + sb.length());
        System.out.println(" cap:" + sb.capacity());
        System.out.println();
        sb.insert(0, "abcdefghi");
        System.out.println(sb.toString());
        System.out.print("length:" + sb.length());
        System.out.println(" cap:" + sb.capacity());
        System.out.println();
        sb.replace(2, 5, "Hello");
        System.out.println(sb.toString());
        System.out.print("length:" + sb.length());
        System.out.println(" cap:" + sb.capacity());
        System.out.println();
    }
}
```

- Below is the output from the example.

```
length:0 cap:16

123456789
length:9 cap:16

abcdefghi123456789
length:18 cap:34

abHellofghi123456789
length:20 cap:34
```

Naming Conventions

- Now that we have seen some examples of variable names, methods, and classes, it is important to point out that Sun Microsystems has established some de facto standards on naming conventions.
  - It is strongly suggested that you stick with these conventions since experience has shown that doing so saves time and effort in the end.

- Variable names should begin with a lowercase letter of the alphabet followed by letters, digits, underscores(_), and dollar signs ($).
  - Words within the name of the variable should have their first letter capitalized. Here are some examples.

```
data     myName     dayOfTheWeek     employeeName
```

- Method names should obey the same rules as variables.

```
sqrt     toString     actionPerformed     readLine
```

- Class names should obey the same rules as variables, except the first letter of a class should be capitalized.

```
StringBuffer     String     Loan     Button
```

- Constants should be capitalized and contain the underscore(_) between "words."

```
TAX_RATE     PI
```

The Date Class

- We have been referring to Java classes by their class name only. For example, we have referred to the String class and the StringBuffer class.

- In reality, since most classes are contained in a package, the package name is also part of the name for a class. Therefore, we speak of the fully qualified name for a class.
  - For example, the fully qualified name of the:
    - String class is java.lang.String; and
    - StringBuffer class is java.lang.StringBuffer.

- When your code is compiled, each class encountered that is not a part of the java.lang package must have its fully qualified name specified in some way in your code.

- The Date class is the first class that we examine, which is not a part of the java.lang package.
  - The Date class exists in the java.util package so you must do only one of the following.
    - Use the fully qualified name of the class in the body of the code as shown here.

```
java.util.Date today = new java.util.Date();
```

    - Use an import statement at the top of the source file allowing the class to be referenced by its short name, as demonstrated in the example on the next page.

The `Date` Class

- The example below demonstrates the use of an `import` statement to use a class that is not part of the `java.lang` package.

`DateTest.java`

```
package examples.ooprogramming;
import java.util.Date;
public class DateTest {
    public static void main(String args[]) {
        Date now = new Date();
        System.out.println(now.toString());
        System.out.println(now);

        System.out.print("Day of the Week: ");
        System.out.println(now.getDay());

        System.out.print("Day of the Month: ");
        System.out.println(now.getDate());

        System.out.print("Month: ");
        System.out.println(now.getMonth());

        System.out.print("Year: ");
        System.out.println(now.getYear());

        System.out.print("Time: ");
        System.out.println(now.getTime());
    }
}
```

  - Note that the `Date` constructor creates a `Date` object with today's date in it.
  - In the first `println` above, the `toString` method was called on the `Date` object to convert it to a `String` for printing purposes.
  - When a reference type is given as an argument to the `println` method, the `toString` method is called automatically, making the second `println` above equivalent to the first.

The `import` Statement

- You may have noticed the `import` statement for the `java.util.Date` class in the previous example.

- Sometimes there exists more than one class from the same package that you wish to import.

```
import java.util.Date;
import java.util.StringTokenizer;
```

- There is a way of telling the Java compiler that you wish to import all of the classes in the same package.

```
import java.util.*;
```

- There are tradeoffs about which style to use.
  - The short style is simpler to code
  - The longer style is more revealing.

- The choice of style has no effect on execution speed of the program.

Deprecation

- When you look at the documentation for many of the methods in the `Date` class, you will see that they have been deprecated.
  - This means that at some future point, these methods will no longer be allowed.
  - Nobody seems to know when that date will arrive, but it is a good idea to follow the fixes as indicated in the documentation.
- You may also notice that when you compile the code on the previous page, you will get a deprecation message.
  - The actual output of the compiler depends on the version of the JDK you are using.
  - The `-deprecation` or `-Xlint:deprecation` options to the compiler can be used to see more information regarding the deprecated methods.

The `StringTokenizer` Class

- Another useful class in the `java.util` package is the `StringTokenizer` class.
    - This is used when you need to separate a long string into component words. Below is an example of how you might use this class.

`StringTokenizerTest.java`

```
package examples.ooprogramming;
import java.util.StringTokenizer;
public class StringTokenizerTest {
    public static void main(String args[]) {
        String text = "Mon Tue Wed Thu Fri Sat Sun";
        StringTokenizer st;
        st = new StringTokenizer(text);
        while(st.hasMoreTokens()) {
            System.out.println(st.nextToken());
        }

        System.out.println("--------------");

        text = "Data,More Data-StillMoreData";
        st = new StringTokenizer(text, ",-");
        int numTokens = st.countTokens();
        for(int i = 0; i < numTokens; i++){
            System.out.println(st.nextToken());
        }
    }
}
```

- Notice that the `StringTokenizer` has several constructors, each of which takes a different set of arguments (another example of method overloading).
    - The constructor that takes two `String` objects uses each of the characters in the second `String` as individual delimiters to separate the tokens in the first `String`.

The `DecimalFormat` Class

- Finally, we show some methods of the `DecimalFormat` class from the `java.text` package.
  - As the name suggests, this class is useful in formatting decimal values.
  - The example below demonstrates several formats for both a `double` and an `int`.

DecimalFormatTest.java

```
package examples.ooprogramming;
import java.text.DecimalFormat;
public class DecimalFormatTest {
    public static void main(String args[]) {
        double value1 = 10000/6.0;
        int value2 = 25;
        DecimalFormat dfA =
            new DecimalFormat(".##");
        DecimalFormat dfB =
            new DecimalFormat(".00");
        DecimalFormat dfC =
            new DecimalFormat(".###");
        DecimalFormat dfD =
            new DecimalFormat("##,###.##");
        DecimalFormat dfE =
            new DecimalFormat("00,000.##");

        System.out.println(dfA.format(value1));
        System.out.println(dfB.format(value1));
        System.out.println(dfC.format(value1));
        System.out.println(dfD.format(value1));
        System.out.println(dfE.format(value1));

        System.out.println(dfA.format(value2));
        System.out.println(dfB.format(value2));
        System.out.println(dfC.format(value2));
        System.out.println(dfD.format(value2));
        System.out.println(dfE.format(value2));

    }
}
```

Exercises

1. Create an application that loops through a `String` object and copies the characters into a `StringBuffer` object, stripping all vowels in the process.

   - The code below can be used as a starting point.

```
public class StripVowels {
    public static void main(String args[]) {
        String input = new String("Now is the time");
        StringBuffer output = new StringBuffer();

        // Fill in missing code here

        // Print the results
        System.out.println(output);
    }
}
```

   - The expected output should be similar to what is shown below.

```
java solutions.ooprogramming.StripVowels
Nw s th tm
```

2. Create a class named `Person` that represents a person's first name, last name and age.

   - The class should have the following constructors.

```
public Person(String first, String last, int age){}
public Person(String fullName, int age){}
```

   - The class should have the following methods.

```
public String getFirstName(){}
public String getLastName(){}
public String getFullName(){}
public int getAge(){}
public String toString(){}
```

   - Create another class that has a `main` method that creates several `Person` objects and tests the methods defined in the class.

Exercises

3. Write a class named `PaddedString` that will represent a new data type that can be padded with whitespace.

- It is recommended that the data in the class be stored in a `StringBuffer` to allow it to be manipulated easily.

```
StringBuffer data;
```

- The constructors for this class should be able to handle either a `String`, an `int`, or a `double` as the parameter, even though it will be stored in the `StringBuffer`.

```
     public PaddedString(String input){ }
     public PaddedString(double input){ }
     public PaddedString(int input){ }
```

- The class should have the following methods.

```
// removes leading and trailing whitespace
public void trimBlanks(){ }

// adds whitespace to the left until the
// overall length is equal to fieldWidth
public void padLeft(int fieldWidth){ }

// adds whitespace to the right until the
// overall length is equal to fieldWidth
public void padRight(int fieldWidth){ }

// Methods to replace values after construction
public void replace(String input){ }
public void replace(double input){ }
public void replace(int input){ }

// Return the data as a String object
public String toString(){ }

// return the number of characters in the data
public int length(){ }
```

- To test the class defined above, create another class named `PadTest` with a `main` method that creates and manipulates several `PaddedString` objects.

Exercises

4. Write a class named `SimpleDate` that represents a `month`, `day`, and `year`.

- This class should provide constructors that enable a user to create `SimpleDate` objects in the following ways.

```
// user supplies the month, day and year
SimpleDate sd1 = new SimpleDate(12, 31, 2004);

// user only supplies the month and day
// constructor will determine current year
SimpleDate sd2 = new SimpleDate(10,31);

// user only supplies the day
// constructor will determine month & year
SimpleDate sd3 = new SimpleDate(15);

//user does not supply anything
//constructor will determine month, day & year
SimpleDate sd4 = new SimpleDate();
```

- The class should have the following methods.

```
public void setDay(int d){}
public void setMonth(int m){}
public void setYear(int y){}
public int getDay(){}
public int getMonth(){}
public int getYear() {}
public String toString(){}
```

- Finally, you should create an application named `SimpleDateTest,` whose `main` method creates several `SimpleDate` objects and calls several of the available methods on each of the objects.

Chapter 4: Methods

Introduction

- We have already seen that a class defines a new data type.

```
Loan  String  StringBuffer
```

  - Each class encapsulates data and methods.
  - Each method describes an action capable of being performed **on** an object of the class or **by** an object of the class.

- This section concentrates on methods.
  - The general form of a method is:

```
modifier return_type name(parameter_list) {
    // body of method
}
```

  - The `modifier` usually refers to an access level.
    - We will assume `public` for now and revisit this topic later.
  - The `return_type` refers to the data type of the value that is returned by the method or `void` if no value is returned.
    - Legitimate values that can be returned by a method include Java primitive types or Java reference types.
  - The `parameter_list` refers to the list of variables (and their types) that will be receiving values from the call to this function. This data is necessary so that the method can do its job.
    - For example, the `setRate` method of the `Loan` class needs to be passed a value so the new rate can be set for this Loan.

Method Signatures

- In order to invoke a method properly, one must know the:
  - name of the method;
  - parameter list of the method; and
  - return type of the method.
- The first two items above are known as the **signature** of the method.
  - The Java docs provide signatures for all methods of the SDK.
- In Java, all methods must be contained inside of a class definition.

Arguments and Parameters

- When a method is defined, it must specify the parameters needed to perform its task.

    - Recall the `setRate` method of the `Loan` class from the previous chapter.

        ```
        public void setRate(double r){
            // body of method
        }
        ```

        - The method defines a single parameter of type `double` that it needs to perform its task.
        - The choice of the variable name `r` is arbitrary.

- To invoke a method, the proper arguments must be supplied.

    - The arguments sent to the method must match the parameter list defined for the method in both number and data type.

        ```
        double defaultAPR = 8.0;
        Loan myLoan = new Loan();
        myLoan.setRate(defaultAPR);
        ```

- The above code is an example of passing a primitive to a method.

    - A copy of the value `defaultApr` is passed to the parameter `r.` This is called a **pass-by-value.**

    - Now we need to discuss what happens when a reference variable is passed as a parameter to a method.

        - This is called a **pass-by-reference**, since it is a copy of the reference (not the actual object) that is passed.

Passing Objects to Methods

- To demonstrate the passing of objects to methods, the `Loan` class will be enhanced by adding a method allowing one person to assume the loan of another person.
  - The revised version of the `Loan` class is shown below.

Loan.java

```
package examples.methods;
public class Loan {
    String name;
    double amount, rate;
    int years;

    public Loan(String n, double a, double r, int y){
        name = n;
        amount = a;
        rate = r;
        years = y;
    }
    public void assume(Loan source){
        double temp = source.amount;
        amount = amount + temp;
        source.amount = 0.0;
    }
    public String toString(){
        StringBuffer sb = new StringBuffer();
        sb.append(name);
        sb.append(", ");
        sb.append(amount);
        sb.append(", ");
        sb.append(rate);
        sb.append(", ");
        sb.append(years);
        return sb.toString();
    }
    // remainder of methods not shown but
    // exist in source file
}
```

  - `toString` method is added to simplify printing of a `Loan` object.

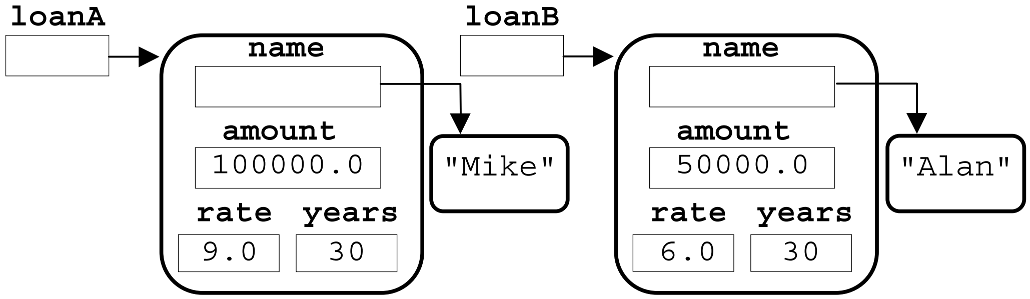
Passing Objects to Methods

- The example below creates several `Loan` objects and calls the `assume` method.

AssumeLoan.java

```
package examples.methods;
public class AssumeLoan {
    public static void main(String args[]) {
        Loan loanA =
            new Loan("Mike", 100000.0, 9.0, 30);
        Loan loanB =
            new Loan("Alan", 50000.0, 6.0, 30);
        System.out.println("Before: " + loanA);
        System.out.println("Before: " + loanB);
        loanA.assume(loanB);
        System.out.println("After: " + loanA);
        System.out.println("After: " + loanB);
    }
}
```

- The variable `loanA` is a reference to a `Loan` object and the variable `loanB` is a reference to a different `Loan` object, as shown below.



- When the `assume` method is invoked on `loanA` with the argument `loanB`, a copy of the reference to `loanB` (not the data to which it refers) is passed to the parameter `source` of the `assume` method.

- Therefore, inside the `assume` method, `source` refers to the same object which `loanB` refers to, as shown on the next page.

Passing Objects to Methods

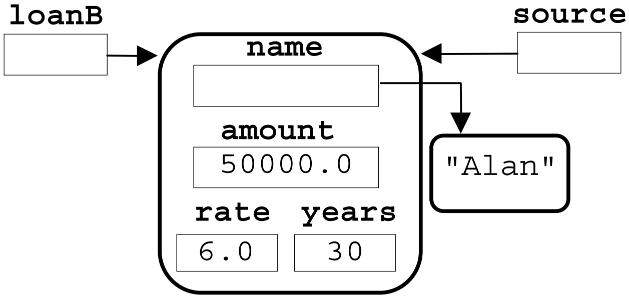
- The `assume` method in `Loan` was called from the `main` method of `AssumeLoan` as follows.

```
loanA.assume(loanB);
```

  - The `assume` method itself is shown below.

```
public void assume(Loan source){
     double temp = source.getAmount();
     amount = amount + temp;
     source.setAmount(0.0);
}
```

  - Inside of the `assume` method, `source` is referencing the data in the object to which `loanB` refers.



  - The unqualified `amount` variable is the value for the `amount` of the object the method was called upon (**the host object**).

Method Overloading

- You may have noticed that any of the primitive Java types can be printed with the `println` method.

```
double x = 50;
int y = 20;
System.out.println(x);
System.out.println(y);
```

  - The `println` methods used above have different signatures.

```
public void println(int val);
public void println(double val);
```

- This is an example of **method overloading** (more than one method has the same name), which is allowed in Java as long as the parameter lists are different.

- You will often see constructors overloaded. A few from the `StringBuffer` class are shown below.

```
public StringBuffer();
public StringBuffer(int length);
public StringBuffer(String text);
```

- Any method can be overloaded. For example, there might be two `setRate` methods.

```
public void setRate(double r){
    // body of method
}
public void setRate() {
  // set rate to some default value
}
```

Static Methods

- As we have seen, many methods are executed on objects, and typically operate on data within an object.

```
Loan myLoan = new Loan("Alan", 100000.0, 8.0, 30);
double r = myLoan.getRate();

StringBuffer sb = new StringBuffer("Boston");
sb.append(" Red Sox");

Date today = new Date();
int year = today.getYear();
```

  - Methods that are executed on an object are called **instance methods**.
    - This is because an object is an instance of a class.
  - The data stored inside an object is called object data or **instance data.**

- However, some methods are not executed on objects and make no use of the data inside of an object.
  - This type of method is called a **static** method.
  - Static methods use the `static` qualifier in their definition.
  - Static methods are viewed as utility functions that perform some type of generic calculation.
  - A static method invocation typically has the name of the class in which it is defined to the left of the dot (`ClassName.method`).

- The example on the next page demonstrates the defining and use of static methods.

Static Methods

- The example below defines a static method named `boxIt` that is called from within the `main` method.

BorderPrinter.java

```
package examples.methods;
public class BorderPrinter {
    public static void main(String args[]){
        String s = "Border Printer";
        BorderPrinter.boxIt(s);
        s = "Place a border around this also";
        boxIt(s);
    }

    public static void boxIt(String data){
        int len = data.length();
        for(int i = 0; i < len + 4; i++){
            System.out.print('*');
        }
        System.out.println();
        System.out.println("* " + data + " *");
        for(int i = 0; i < len + 4; i++){
            System.out.print('*');
        }
        System.out.println('\n');
    }
}
```

  - Neither `main` nor `boxIt` are called on objects.
    - Each is defined as a static method using the keyword `static.`
    - `boxIt` is called from within the class where it is defined and therefore, it is permissible to exclude the class name as a qualifier.

- The example on the next page expands upon the use of static methods.

Static Methods

- The example below defines a static method named `titleCase` and also calls the static method `boxIt` from the previous example.

MessagePrinter.java

```
package examples.methods;
import java.util.*;
public class MessagePrinter {
    public static void main(String args[]){
        String s = "Static Method Demo";
        BorderPrinter.boxIt(s);
        String msg = "message to be converted";
        titleCase(msg);
        msg = "another simple message";
        titleCase(msg);
    }
    public static void titleCase(String s){
        StringTokenizer st = new StringTokenizer(s);
        String tmp;
        String first;
        while(st.hasMoreTokens()){
            tmp = st.nextToken();
            first = tmp.substring(0,1).toUpperCase();
            System.out.print(first);
            System.out.print(tmp.substring(1) + " ");
        }
        System.out.println();
    }
}
```

- Since the `boxIt` method is not defined in the same class as where it is being called, it is required that it be qualified with the class name in order for the compiler to know where to find the definition.

```
BorderPrinter.boxIt(s);
```

- The `titleCase` method does not have to be qualified since it is defined in the same class from which it is being called.

The `Math` Class

- This class is a little different from other classes for several reasons.
    - There can be no objects defined.
    - All of the methods in this class are static.

- It is expected that the methods in the `Math` class will be used in a wide variety of applications, which may need some mathematical functionality.
    - These methods are to be viewed as utility functions and are not executed on objects.
    - The example below tests several methods in the `Math` class.

MathTest.java

```java
package examples.methods;
public class MathTest {
    public static void main(String args[]) {
        double y;

        // square root of a number
        y = Math.sqrt(25.0);
        System.out.println("Sqrt:   " + y);

        // 3 raised to the 4th power
        y = Math.pow(3.0, 4.0);
        System.out.println("Powers:    " + y);

        // random number between 0 and 1
        y = Math.random();
        System.out.println("Random: " + y);

        // Round a number to nearest long
        long z;
        z = Math.round(-2.6);
        System.out.println("Rounding: " + z);
    }
}
```

The System Class

- The System class is another useful class that consists mainly of static methods.

- The exit method terminates the currently running JVM.
  - The int argument passed to this method serves as a status code.
    - By convention, a nonzero status code indicates abnormal termination.

- The currentTimeMillis method returns a long which is the number of milliseconds that have elapsed since the beginning of Jan 1, 1970.
  - The Date class has a constructor that takes a long to create a Date object.
  - Calling currentTimeMillis twice within a program can provide the ability to determine the length of time elapsed, by calculating the difference between the two values.

- The example on the next page demonstrates calling some static methods from the System class.

The System Class

SystemStuff.java

```
package examples.methods;
import java.util.Date;
public class SystemStuff {
    public static void main(String args[]) {
        long t1 = System.currentTimeMillis();
        System.out.println("ms = " + t1);

        Date today = new Date(t1);
        System.out.println("Today is: " + today);

        if(Math.random() < .5){
            String s = "JVM terminating early";
            System.out.println(s);
            System.exit(1);
        }

        long day = 1000*60*60*24;
        Date tomorrow = new Date(t1 + day);
        System.out.println("Tomorrow: " + tomorrow);

        long t2 = System.currentTimeMillis();
        System.out.println("# of ms: " + (t2-t1));

        System.out.println("JVM terminating");
    }
}
```

- The example above will terminate in one of two ways, depending on the random value obtained inside the `if` statement.

Wrapper Classes

- There is a set of classes in Java which provide object versions of the primitive data types.
  - These classes are collectively referred to as **wrapper** classes.
  - The class names are based on the primitive data type names and are listed below.

    ```
    Byte     Double    Float     Integer
    Long     Short     Boolean   Character
    ```

  - The benefit of wrapping a primitive inside an object will be studied in a later chapter.

- The wrapper classes also contain many static methods for converting a primitive to a `String` or a `String` to a primitive.
  - Several examples of these static methods are shown below.
    - Converting a `String` to an `int`

      ```
      int val = Integer.parseInt("1234");
      ```

    - Converting an `int` to a `String`

      ```
      String s = Integer.toString(1234);
      ```

    - Converting a `String` to a `double`

      ```
      double val = Double.parseDouble("12.34");
      ```

    - Determining the type of a particular character

      ```
      String text = "abc ABC 123";
      for(int i = 0; i < text.length(); i++){
          char c = text.charAt(i);
          System.out.println(Character.isLowerCase(c));
          System.out.println(Character.isWhitespace(c));
          System.out.println(Character.isDigit(c));
      }
      ```

Exercises

1. Write an application that generates random numbers.

   - Generate at least 10,000 numbers, keeping two counts:
     - those that are greater than .5; and
     - those that are less than or equal to .5.
   - Print the results of the two counts.
   - After 10,000 numbers have been generated, continue to generate random numbers and update the two counts, until one of the two conditions below is met.
     - The two counts are equal
     - 1,000,000 numbers have been generated
   - When either condition is met:
     - print out the time it took to run the program using the `currentTimeMillis` method from the `System` class; and
     - terminate the program using the `exit` method of the `System` class, passing an argument of `1` to the `exit` method if the counts are equal or an argument of `2` if 1,000,000 is reached.

2. Write a class named `Count`, which contains static methods that determine the number of characters of certain types held in a `String` or `StringBuffer`.

   - The class should not have any constructors but should have the following static methods.

```
public static int digits(String text){}
public static int whitespace(String text){}
public static int digits(StringBuffer text){}
public static int whitespace(StringBuffer text){}
```

   - This class can be used as a utility class similar to the `Math` class.

Exercises

3. Modify the `SimpleDate` class created as an exercise in the previous chapter so that it allows a user to construct a `SimpleDate` by passing a `String` of the form `"m/d/yyyy."`
   - A copy of `SimpleDate.java` can be found in the starters directory of this chapter if necessary (but it is recommended that you use the one you developed previously).
4. Make the following changes to a copy of the `Loan` class that can be found in the starters directory for this chapter.
   - Add a method to compute the amount of the monthly payment based on the formula below.

```
M = P * ( J / ( 1 - (( 1 + J) ** - N)))

where

M = Monthly payment
P = Initial Loan Amount
I = Interest rate
J = I / (12 x 100) Monthly Interest in decimal form
N = Number of months over which loan is amortized
```

   - Test your program by computing the monthly payments for a $200000 mortgage at 7.5% over 30 years.
     - The result should be `1398.43.`

This Page Intentionally Left Blank

Chapter 5:
Arrays

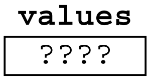
Introduction

- An array is an ordered collection of data items all of whose types are the same.

- In Java, an array is an object, so it must have a reference pointing to it.
  - The reference can be declared in either of the following ways.
    - We will use an array of `int` data types for our examples, but any data type would do.

```
int values[];
int [] values;
```

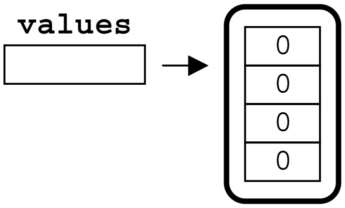
- Either of the above defines a reference to an array of integers. However, there is no array yet.
  - The storage for the above code looks like what is shown below.



  - To make the variable `values` reference an array, we must construct the array object using the `new` operator.

```
values = new int[4];
```

    - Now the storage changes to the following.



- The declaration and initialization could be combined as:

```
int values[] = new int[4];
```

Processing Arrays

- Since an array is an object, it has certain properties.
  - The `length` property is provided for all arrays.
    - `length` is a property (not a method), so it is not followed by a set of parenthesis.

      ```
      int x = values.length;   //length of array
      ```

- Arrays are typically processed with loops.
  - The code below demonstrates this by setting each element of the array and then by totaling these elements.

    ```
    int sum = 0, i;
    int values[] = new int[5];
    for(i = 0; i < values.length; i++)
        values[i] = i;
    for(i = 0; i < values.length; i++)
        sum += values[i];
    ```

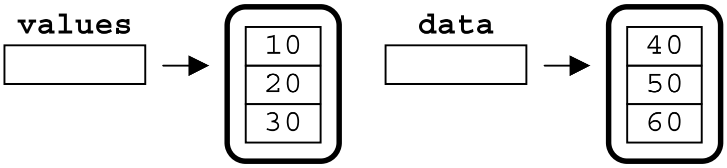
- An array can also be created and initialized with the following syntax.

```
int values [] = {0, 1, 2, 3, 4};
```

Copying Arrays

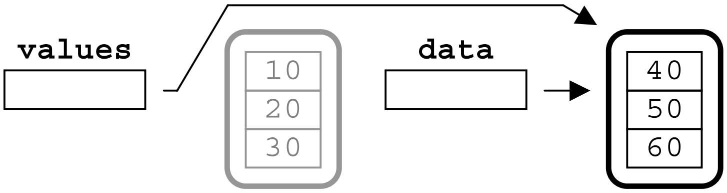
- Since an array is a reference type, you need to be careful when you are copying one array to another.
  - Examine the following code and accompanying pictures.

```
int values[] = {10, 20, 30};
int data[]   = {40, 50, 60};
```



  - When you attempt to copy one to the other using an assignment operator, only the reference is copied.

```
values = data;
```



    - This is not a true copy in that only one set of values exists, and any change to the object referenced by `values` will result in a change to the object referenced by `data`.
    - Nothing is referencing the object containing the values {10, 20, 30} and, as such, this object and the memory it occupies is available for garbage collection.

- If a true copy of an array of primitives is desired, Java provides a static method named `arraycopy` in the `System` class as shown on the next page.

Copying Arrays

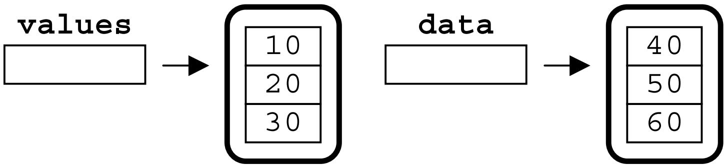
- Using the `System.arraycopy` method to copy one array to another is detailed below.

```
System.arraycopy(src, srcPos, dest, destPos, length);
```

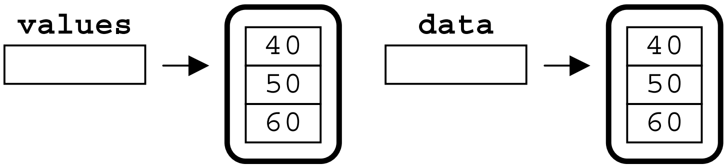
  - The parameters of the `arraycopy` method follow.
    - `src` - source array (copy from)
    - `srcPos` - starting index in source array
    - `dest` - destination array (copy to)
    - `destPos` - starting index in destination array
    - `length` - number of elements to be copied

- The code below demonstrates the use of the `arraycopy` as a way of setting `values` equal to an actual copy of `data.`

```
int values[] = {10, 20, 30};
int data[]   = {40, 50, 60};
```



```
System.arraycopy(data, 0, values, 0, 3);
```



  - Below is an example of copying only part of an array into another using the `arraycopy` method.

```
System.arraycopy(data, 0, values, 1, 2);
```

Passing Arrays to Methods

- The example below demonstrates what happens when an array is passed as an argument to a method.
  - Since an array is a reference type, only the reference is passed, rather than the object itself.
  - This results in the method having direct access to all the elements of the array.

PassAnArray.java

```
package examples.arrays;
public class PassAnArray {
    public static void main(String args[]) {
        int [] x = { 1, 2, 3, 4, 5};
        print(x);
        multiply(x, 3);
        print(x);
    }
    public static void multiply(int [] p, int val){
        for(int i = 0; i < p.length; i++)
            p[i] *= val;
    }
    public static void print(int [] p) {
        for(int i = 0; i < p.length; i++)
            System.out.print(p[i] + " ");
        System.out.println();
    }
}
```

  - Inside the `print` and `multiply` methods above, the reference `p` refers to the same object that is referenced by `x`.

- Now that we have demonstrated several issues of dealing with an array of primitives, there are a few other details to investigate when dealing with an array of reference types.

- The example on the next page defines a `Point` class that will be used to investigate arrays of objects.

Arrays of Objects

- The examples below define a `Point` class and an application that creates an array of `Point` objects.

Point.java

```
package examples.arrays;
public class  Point {
    int xc, yc;

    public Point(int x, int y) {
        xc = x;
        yc = y;
    }
    public int getXc() {
        return xc;
    }
    public int getYc() {
        return yc;
    }
    public String toString() {
        return xc + "," + yc;
    }
}
```

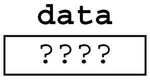
PointTest.java

```
package examples.arrays;
public class PointTest {
    public static void main(String args[]) {
        Point data[];
        data = new Point[3];
        data[0] = new Point(2,3);
        data[1] = new Point(4,5);
        data[2] = new Point(6,7);
        for (int i = 0; i < data.length; i++)
            System.out.println(data[i]);
    }
}
```

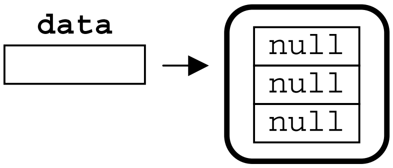
Arrays of Objects

- The diagrams below demonstrate the memory allocation for each object as the main method is executed.

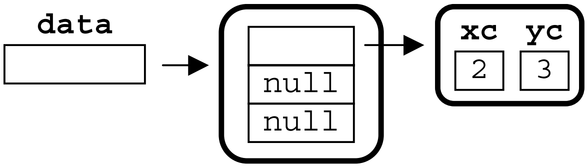
```
Point data[];
```



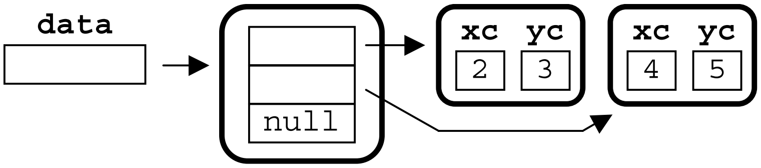
```
data = new Point[3];
```



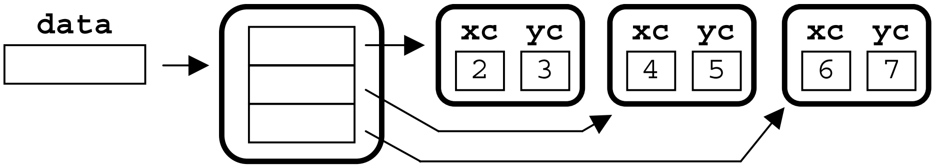
```
data[0] = new Point(2,3);
```



```
data[1] = new Point(4,5);
```



```
data[2] = new Point(6,7);
```



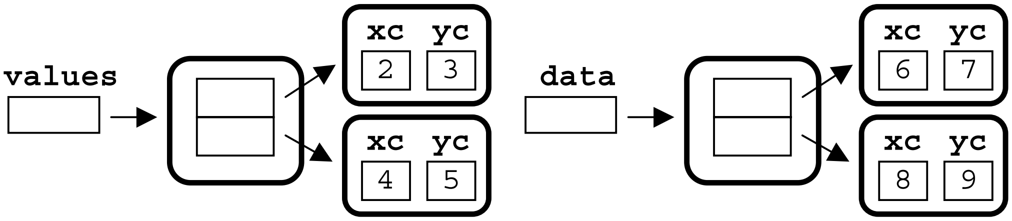
- The code could also have been written as follows.

```
Point data[] = {new Point(2,3),
                new Point(4,5),
                new Point(6,7)};
```

Arrays of Objects

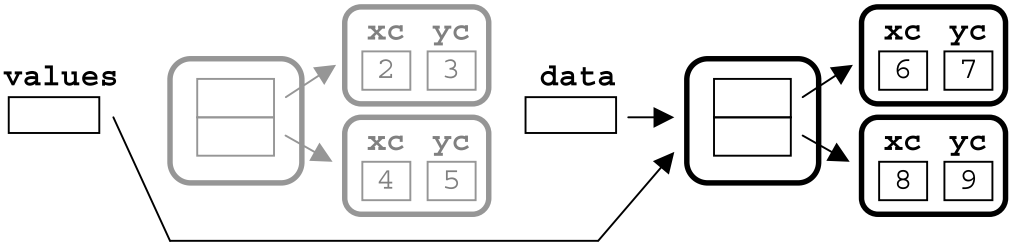
- As with arrays of primitives, you should be careful when copying arrays of objects.
  - The two arrays shown below will be used to demonstrate the details of copying arrays of objects.

```
Point values[] = {new Point(2,3), new Point(4,5)};
Point data[]   = {new Point(6,7), new Point(8,9)};
```



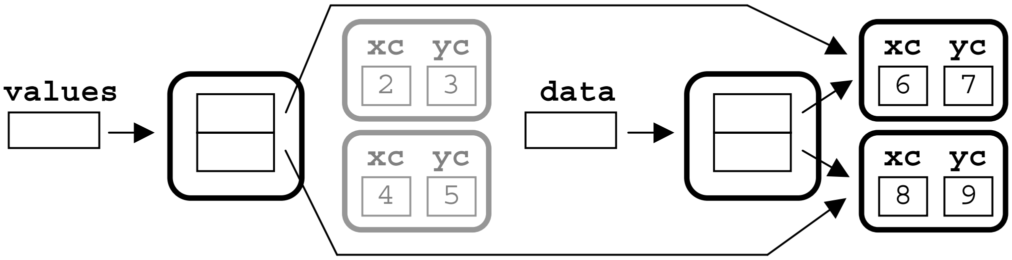
  - Assigning `values` equal to `data` merely results in the two references pointing to the same array of references as shown below.

```
values = data;
```



  - Even if `System.arraycopy` is used, only the references in the array are copied.

```
System.arraycopy(data, 0, values, 0, 2);
```



The Arrays Class

- The `Arrays` class is another utility class in the `java.util` package containing several overloaded static methods for manipulating an array of any data type.
  - The example below demonstrates several of these methods.

ManipulateArrays.java

```
package examples.arrays;
import java.util.Arrays;
public class ManipulateArrays {
    public static void main(String args[]){
        int x [] = {3, 7, 1, 9, 2, 10};
        String s [] = {"Mike", "Alan", "Susan"};
        printArray(x);
        printArray(s);
        System.out.println("\nSorted Order");
        Arrays.sort(x);
        Arrays.sort(s);
        printArray(x);
        printArray(s);
        System.out.print("Filling an array:    ");
        Arrays.fill(s, "NotUsed");
        printArray(s);
    }
    public static void printArray(int a[]){
        for(int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();
    }
    public static void printArray(String s[]){
        for(int i = 0; i < s.length; i++)
            System.out.print(s[i] + " ");
        System.out.println();
    }
}
```

Command Line Arguments

- In Java, command line arguments are passed to the `main` method of the application as an array of `String` objects. This is demonstrated in the following example.

`Arguments.java`

```java
package examples.arrays;
public class Arguments {
    public static void main(String args[]) {
        if ( args.length == 0 ){
            System.out.println("Need arguments");
            System.exit(1);
        }
        System.out.println("FORWARD");
        for (int i = 0; i < args.length; i++)
            System.out.println(args[i]);
        System.out.println("\nBACKWARD");
        for (int i = args.length - 1; i >= 0; i--)
            System.out.println(args[i]);
    }
}
```

- When a Java program is executing, all array references are checked to see that they are within the bounds of the array.
  - An array subscript less than zero or greater than or equal to the upper bound of the array will cause a run time error.
  - Some of these errors can be subtle as shown below.

`OffByOne.java`

```java
package examples.arrays;
public class OffByOne {
    public static void main(String args[]) {
        for (int i = 0; i <= args.length; i++)
            System.out.println(args[i]);
    }
}
```

- If the application above is run, the JVM will throw an exception at runtime.

Multidimensional Arrays

- Arrays can be of any number of dimensions.
  - Of course, the more dimensions an array has, the less common they are in programming problems.
  - Two and three-dimensional arrays are extremely common.

- Below is an example of a two-dimensional array.
  - This reference type is best thought of as being composed of several one-dimensional arrays.
  - Keep in mind that a one-dimensional array is itself a reference type.

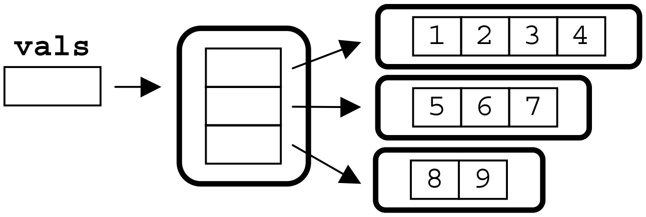
```
int data[][] = new int[3][3];
```

  - You can reference an entry in the array above as follows.

```
data[0][0] = 1;
```

- If you wanted a two-dimensional array, which was not rectangular, you could build it as follows.

```
int vals[][] = new int[3][];
vals[0] = new int[4];
vals[1] = new int[3];
vals[2] = new int[2];
int count = 1;
for (int i = 0; i < vals.length; i++)
    for (int j = 0; j < vals[i].length; j++)
        vals[i][j] = count++;
```



Exercises

1. Write a class named `StringOrganizer`, which encapsulates an array of `String` objects and defines several methods that can be performed on the data.

   - The class should define the following.

   ```
   // instance variables
   String data [];

   // constructors
   public StringOrganizer (String [] args){}

   // instance methods
   public void reverse(){}
   public void ascendingSort(){}
   public void descendingSort(){}
   public String getString(int index){}
   public String toString(){}
   ```

   - Create a program whose main method creates a new `StringOrganizer` object with the data received from the command line and tests the methods defined in the `StringOrganizer` class.

2. Create a program named `TemperatureConverter`, which takes three values from the command line: a beginning temp (celsius), an ending temp (celsius), and an increment value.

   - The program should produce a table of temperature conversions as shown below.

   ```
   java solutions.arrays.TemperatureConverter  0 30 10
   CELSIUS FAHRENHEIT
   0       32.0
   10      50.0
   20      68.0
   30      86.0
   ```

   - The equation for converting Celsius to Fahrenheit is:

   ```
   f = 1.8 * c + 32
   ```

Exercises

3. Write a class named `Assets,` which tracks your favorite items.

   - Objects of this class should contain a few arrays so that when users add an item, the object can store its name and its value.
   - Your class should have the following.

```
// instance variables
String names[];
double values[];
int size, capacity;

// constructor
public Assets(int maxSize){}

// methods

// add an item and its value to this object
public void addElement(String item, double itemVal){}

// number of items currently stored
public int size(){}

// number of items capable of storing
public int capacity(){}

// total dollar value of all assets being tracked
public double getTotalValue(){}

// return a String that contains name and value
// of a particular item
public String getItem(int whichItem){}

// table of item names and values
public String toString(){}
```

   - The file named `TestAssets.java` in the starters directory for this chapter is already completed and can be used to test the above methods in your `Assets` class.

Exercises

4. Start with the `Loan` class in the starters directory.

- In this exercise you will add a few arrays to the `Loan` class: `interest`, `principal`, and `balance`.
- These arrays will keep track of:
  - `interest` - The interest due each month
  - `principal` - The principal being paid off each month
  - `balance` - The total balance due after the payment has been made for that month.
- Add a method to produce a table for the first `n` months.

```
public void printTable(int months)
```

- Use the `DecimalFormat` class so that the output values show two decimal digits.
- Add another method that computes the total interest paid over the lifetime of the loan.

```
public double totalInterest()
```

This Page Intentionally Left Blank

Chapter 6: Encapsulation

Introduction

- We have seen many examples of classes in previous chapters.
  - Recall that a class defines a new data type.
- A data type has a representation and a set of operations that realize the behavior of the type.
  - For example, an `int` is a data type. It is represented in memory by 32 bits and a coding scheme that is the binary number system.
- Homegrown data types, like the `Loan` type, have their own representation and operations.
  - The operations for a `Loan` are defined by a set of methods that are encoded within the `Loan` class definition.
- The coupling of data + methods is known as **encapsulation**.
  - This section gives many of the details of classes related to encapsulation.
  - The next chapter gives additional Object-Oriented related information, namely **inheritance** and **polymorphism**.

Constructors

- A **constructor** is a special (possibly overloaded) method that bears the name of the class.
  - Its principal function is initialization, but it can do anything that other methods can do except return a value.
  - Constructors are called automatically when the `new` keyword is used to instantiate an object.
  - Although constructors behave largely like other methods, they cannot have a return type – not even `void`.
  - In the absence of a specific constructor, the compiler will supply a default constructor, which merely allocates space for the object and fills in data members with default values.

- If a class has one or more constructors, then the set of them describes how objects of that class can be constructed.

- It is perfectly legitimate for more than one method or constructor within a class to have the same name as long as their parameter lists are different.
  - This concept is called method overloading.

- The example on the next page revisits the `Point` class to study some of the details related to constructors.

Constructors

- When a constructor is executed, there is an order of events that occurs as detailed below.
  - All object data is set to default values:

`0`	for numeric types
`'\0'`	for char types
`false`	for boolean types
`null`	for reference types

  - If any object data is initialized, it is then set to the initialization value.
  - The body of the constructor is then executed.

- To demonstrate how the above rules apply, the `Point` class studied earlier has been rewritten as shown below.

Point.java

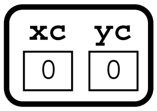
```
package examples.encapsulation;
public class  Point {
    int xc = 1;
    int yc = 2;

    public Point(int x, int y) {
        xc = x;
        yc = y;
    }
    public int getXc() {
        return xc;
    }
    public int getYc() {
        return yc;
    }
    public String toString() {
        return xc + "," + yc;
    }
}
```

Constructors

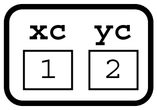
- The diagrams below demonstrate the events that occur when the following statement is executed.

```
Point p = new Point(4, 5);
```

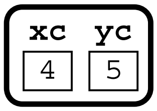
  - First, an object of type `Point` is created, and its instance data is set to 0.



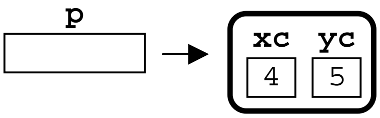
  - Then, the initializers apply and any object data that was declared with an initial value will have that value set.



  - Finally, the body of the constructor is executed.



  - Upon completion of the construction of the object, a reference to the object is stored in the variable `p`.



The `this` Reference

- When a method is called on an object, the object upon which it is called is referred to as the **host object**.
  - The method being called is automatically passed a reference to the host object.
  - Inside the method, the reference is named `this`.

- In other words, the `this` reference is alive only inside instance methods of a class and always references the host object.
  - Whenever instance data is mentioned inside a method of a class, it is as though it were qualified by the `this` reference.
    - In other words, the `Point` class could have been written as shown below.

Point.java

```
package examples.encapsulation;
public class  Point {
    int xc = 1
    int yc = 2;

    public Point(int x, int y) {
        this.xc = x;
        this.yc = y;
    }
    public int getXc() {
        return this.xc;
    }
    public int getYc() {
        return this.yc;
    }
    public String toString() {
        return this.xc + "," + this.yc;
    }
}
```

The `this` Reference

- Sometimes a method of a class needs to call, or can benefit from calling, another method of the same class.
  - Suppose the methods shown below are added to the `Point` class. Each of these methods defines a way to shift a `Point` a certain amount along the x-axis or y-axis.

```
public void shiftX(int x) {
     xc += x;
    // could have been written as this.xc += x
}
public void shiftY(int y) {
     yc += y;
    // could have been written as this.yc += y
}
```

  - With the two methods above added, suppose we now want to define a method named `shift` that takes both the x amount and y amount as parameters to shift the point.
    - The method to be defined can take advantage of the two methods already defined, as shown below.

```
public void shift (int x, int y){
    this.shiftX(x); // call shiftX from same class
    this.shiftY(y); // call shiftY from same class
}
```

    - Since the use of the variable `this` is mostly optional (see note below) the method could be written as shown below.

```
public void shift (int x, int y){
    shiftX(x);
    shiftY(y);
}
```

  - ***Note:*** When a parameter of a method has the same name as an instance variable being referred to within the method - the use of `this` is **required** in order to distinguish between them.

```
public void shiftY(int yc) {
     this.yc += yc;
}
```

The `this` Reference

- There is another use for the `this` reference. If a class has a set of overloaded constructors, you can use the functional form of the `this` reference to call one constructor from another.
    - To demonstrate the functional form of `this`, we will start by defining a `Circle` class with several constructors as shown below.

Circle.java

```
package examples.encapsulation;
public class Circle {
    int xc, yc, radius;
    public Circle(int x, int y, int rad){
        xc = x;
        yc = y;
        radius = rad;
    }
    public Circle(int x, int y) {
        xc = x;
        yc = y;
        radius = 1;
    }
    public Circle(int rad) {
        xc = 0;
        yc = 0;
        radius = rad;
    }
    public Circle() {
        xc = 0;
        yc = 0;
        radius = 1;
    }
    public double calcArea() {
        return Math.PI * radius * radius;
    }
    public String toString() {
        return xc + "," + yc + ": rad = " + radius;
    }
}
```

The this Reference

- Much of the work that each constructor needs to accomplish in the Circle class is similar. The example below demonstrates the use of the functional form of this to call one constructor from another.

```
public Circle(int x, int y, int rad){
    xc = x;
    yc = y;
    radius = rad;
}
public Circle(int x, int y) {
    this(x, y, 1);
}
public Circle(int rad) {
    this(0, 0, rad);
}
public Circle() {
    this(0, 0, 1);
}
```

- When the above technique is used, the functional form of this must be the first statement inside the constructor.

Data Hiding

- The `Circle` class is represented by a pair of integers for the center of the circle and another integer for the radius of the circle.
  - The representation is chosen by the designer(s) of the class.
  - User code should never be written with knowledge of this representation, because if the developer later changes the representation, all user code would fail with the new version of the class.
    - For example, the developer of the `Circle` class may later decide to represent the three integers as an array of integers.
  - If user code can directly access instance data of the `Circle` class, careless errors such as the one below can easily result.

```
Circle c1 = new Circle();
c1.radius = -5;
```

    - A better solution is to define a `setRadius` method in the `Circle` class.
    - This allows the developer to check for any unacceptable values prior to storing the value in the instance variable of the object.

```
c1.setRadius(-5);
```

- The example on the next page is a revised version of the `Circle` class that, among other things, prevents a user of the class from setting the radius to a negative value.

`public` and `private` Members

- The example below shows the changes made to the `Circle` class pertaining to hidden data.

```
package examples.encapsulation;
public class Circle {
    private int xc, yc, radius;
    public Circle(int x, int y, int rad){
        xc = x;
        yc = y;
        if(rad < 0){
            print("Bad radius: " + rad);
            print("Default value of 1 being used");
            rad = 1;
        }
        radius = rad;
    }
    public void setRadius(int r){
        if(r < 0){
            print("Bad radius: " + r);
            print("radius " + radius + "unchanged" );
            return;
        }
        radius = r;
    }
    private void print(String msg){
        System.out.println(msg);
    }
    // remainder of class not shown
}
```

- With the instance data of the `Circle` class being defined as `private`, only methods within the `Circle` class can access the object data directly.
    - The compiler will complain if any other class attempts to execute code such as the ones after the constructor below.

```
Circle c1 = new Circle();
c1.radius = 5;           // private access violation
c1.xc = 0;               // private access violation
c1.yc = 10;              // private access violation
```

public and private Members

- Methods can also be declared as private.
  - In this case, these methods can only be called from within methods of this same class.
  - For example, the print method defined in the Circle class on the previous page has been declared as a private method. It is intended to be called only by the constructor of the class and the setRadius method.
    - Since the constructor and the setRadius method are both defined in the same class as the print method, both methods are able to call the print method.
    - On the other hand, the main method (or any other method) inside of a class named TestCircle (or any other class) would not be able to call a private method within the Circle class.
- The example on the next page defines a Fraction class that will incorporate all of the topics discussed in this chapter.
  - The class has a set of overloaded constructors that will rely on the functional form of the this reference.
  - The constructor's parameter list will have the same names as the instance variables, requiring the use of the this reference.
  - Instance data will be declared as private .
  - A method to determine the greatest common divisor of two numbers intended to be used internally by methods of the class, will be declared as private.

`public` and `private` Members

Fraction.java

```
package examples.encapsulation;
public class Fraction {
    private int numer, denom;

    public Fraction(int numer, int denom){
        // use of "this" is required here
        this.numer = numer;
        this.denom = denom;
    }
    public Fraction(){
        this(0,1);
    }
    public Fraction multiply(Fraction p) {
        Fraction temp = new Fraction();
        // use of "this" is optional here
        // simply used for clarity
        temp.numer = this.numer * p.numer;
        temp.denom = this.denom * p.denom;
        return temp;
    }
    public String toString() {
        int val = this.gcd(numer, denom);
        return numer/val + "/" + denom/val;
    }
    private int gcd(int top, int bot) {
        int rem;
        rem = top % bot;
        while(rem != 0) {
            top = bot;
            bot = rem;
            rem = top % bot;
        }
        return bot;
    }
}
```

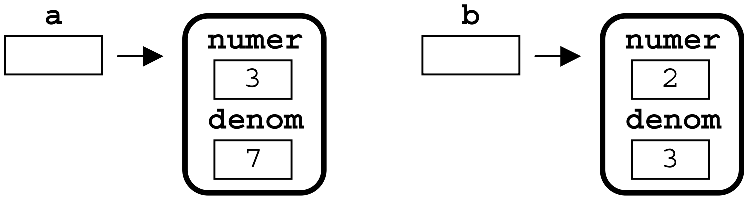
- The example on the next page tests the `Fraction` class above.

`public` and `private` Members

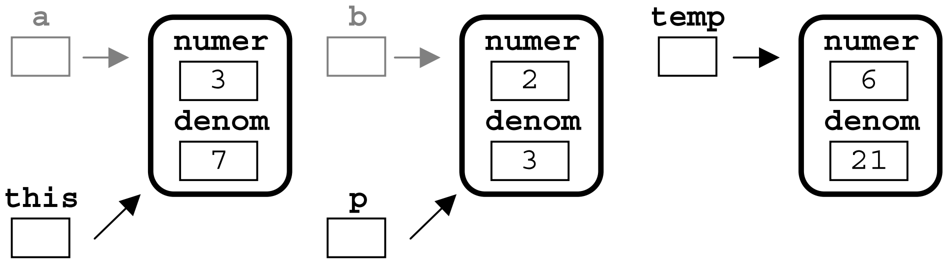
FractionTest.java

```
package examples.encapsulation;
public class FractionTest {
    public static void main(String args[]) {
        Fraction a = new Fraction(3,7);
        Fraction b = new Fraction(2,3);
        Fraction c = a.multiply(b);
        System.out.println(c);
    }
}
```

- The `multiply` method called above deals with the following three `Fraction` objects: the host object (`a`), the argument to the method (`b`), and the object returned (`c`) .
  - The diagram below represents the memory allocated prior to the `multiply` method being called.



  - Inside the `mutiply` method, the pictures changes as shown below.



    - Note that inside of the `multiply` method, the variables `a` and `b` are not available.

Access Levels

- The examples shown so far have demonstrated the difference between `public` and `private` access to both the data and methods of a class.

- In fact, there are four access levels in Java.
  - `public` – access from methods
  - `private` – access limited to class methods
  - `protected` – access limited to class methods, subclass methods, and those methods in the same package
  - **default** – access limited to those methods in the same package

- The most typical situation is to have `private` data members and a set of `public` methods.
  - The set of `public` methods for a class is called the public interface.
  - Data hiding is implemented by using the public interface to access private data.

- The `protected` mechanism will be visited when we study inheritance in the next chapter.

- If a method or instance data has no access level, then default access level is implied (often referred to as package access).
  - This means that only methods in the same package have access to this item.

Composition

- Once a class has been created, it can be used as instance data in other classes.
  - For example, a `Line` may be composed of a pair of x and y coordinates, and a length.
    - But we already have a class named `Point` that is composed of an x-coordinate and a y-coordinate.
    - Therefore, a `Line` can be designed as being composed of two `Point` objects and a `length`.

Line.java

```
package examples.encapsulation;
public class Line {

    private Point p1;
    private Point p2;
    private double length;

    public Line(Point p1, Point p2) {
        this.p1 = p1;
        this.p2 = p2;
        length = distance(p1, p2);
    }
    public Line(int x1, int y1, int x2, int y2) {
        this(new Point(x1,y1), new Point(x2,y2));
    }
    private double distance(Point p1, Point p2) {
        double xd = p1.getXc() - p2.getXc();
        double yd = p1.getYc() - p2.getYc();
        return Math.sqrt(xd * xd + yd * yd);
    }
    public double getLength () {
        return length;
    }
    public String toString() {
        return p1.toString()+ "; " + p2.toString();
    }
}
```

Composition

- Below is an application that tests both of the constructors in the `Line` class and the methods of the class.

LineTest.java

```
package examples.encapsulation;
public class TestLine {
    public static void main(String args[]) {
        Point p1 = new Point(0,0);
        Point p2 = new Point(3,4);
        Line lineA = new Line(p1,p2);
        System.out.println("Line A: " + lineA);
        System.out.println(lineA.getLength());

        Line lineB = new Line(0,0,6,8);
        System.out.println("Line B: " + lineB);
        System.out.println(lineB.getLength());
    }
}
```

- The output from running the above application is shown below.

```
java examples.encapsulation.LineTest
Line A: 0,0; 3,4
5.0
Line B: 0,0; 6,8
10.0
```

Static Data Members

- To now, we have seen the difference between static and instance methods.

- Now we look at the difference between class data and instance data.

- When an object is created, the data that it encapsulates is called **instance** or **object** data.

- There are occasions when a class needs to share data among all objects of the class.
  - This sharable data is not part of each object, but rather it is part of the class and is accessible by methods of the class.
  - Some uses of shared data might be as follows.
    - We may want to track the number of `Point` objects in a program.
    - We may want to know the name of the borrower with the largest `Loan`.
    - We might want to assign each borrower a unique account number.

- Class data in Java is accomplished using the `static` key word. An example follows on the next page.

Static Data Members

Account.java

```
package examples.encapsulation;
public class Account {
    private static int counter = 1000;
    private String name;
    private int accountNum;
    public Account(String n) {
        name = n;
        accountNum  = counter++;
    }
    public String toString() {
        return name + " has account # " + accountNum;
    }
    public static int nextNumber() {
        return counter;
    }
}
```

AccountTest.java

```
package examples.encapsulation;
public class AccountTest {
    public static void main(String args[]) {
        System.out.print("Next # is ");
        System.out.println(Account.nextNumber());
        Account workers [] = { new Account("Mike"),
                               new Account("Susan"),
                               new Account("Alan") };
        for (int i = 0; i < workers.length; i++)
            System.out.println(workers[i]);
        System.out.print("Next # is ");
        System.out.println(Account.nextNumber());
    }
}
```

- The output from the above application is shown below.

```
java examples.encapsulation.AccountTest
Next # is 1000
Mike has account # 1000
Susan has account # 1001
Alan has account # 1002
Next # is 1003
```

Exercises

1. Make the following modifications to the `Fraction` class used earlier in this chapter.

   - Prevent a user from being able to instantiate a `Fraction` object with a denominator of zero.

   - Add the following methods to the class.

   ```
   public Fraction add(Fraction f) {}
   public Fraction subtract(Fraction f) {}
   public Fraction divide(Fraction f) {}
   ```

2. Add the following methods to the `SimpleDate` class completed as an exercise in an earlier chapter.

   - The code below can be found in the starters directory of this chapter in a file named `SimpleDateAdditions.txt`.

   ```
   public String getMonthAsString(){}
   public int getDayOfYear(){}
   public int getDaysLeftInYear(){}
   ```

     - The following static arrays might be helpful.

   ```
   private static int months[] = {31, 28, 31, 30, 31, 30,
        31, 31, 30, 31, 30, 31};
   private static String names[]=
       {"January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"};
   ```

     - The following method determines if the year is a leap year.

   ```
   public boolean isLeapYear(){
     // assuming instance variable is named "year"
     return year % 400 == 0 ||
           ((year % 4 == 0) && (year % 100 != 0))
   }
   ```

   - A test application named `SimpleDateTester.java` can be found in the starters directory for this chapter.

Exercises

3. Write a class named `IDGenerator` that hands out a new id each time its `issueNext` method is called.

   - This class should be able to be used in various scenarios.
     - It may be used to generate employee ids of the form:
       `emp101 emp102 emp103 ...`
     - It may be used to generate account numbers such as:
       `acc1000 acc1001 acc2002`
     - Or to generate a generic id such as:
       `id500 id501 id502`
   - Since this class can be used in various scenarios, it will maintain a `String` for the prefix to be used.
   - The class should also have an `int` that maintains the next number that will be used whenever `issueNext` is called.
     - The `issueNext` method will return the actual id as the prefix and number concatenated together as a `String.`
   - The class should have an overloaded set of constructors including one that:
     - takes no parameters and simply relies on a default prefix and starting number;
     - permits a `String` to be as the prefix and relies on a default starting number; and
     - will take a `String` as the prefix and an `int` as the starting value.
   - The starter code shown on the following page can be found in the starters directory for this chapter.

Exercises

IDGenerator.java

```
package starters.encapsulation;
public class IDGenerator{
    private String prefix;
    private int number;

    // supplies a default prefix
    // supplies a default starting number
    public IDGenerator() {}

    // user supplies prefix
    // supplies a default starting number
    public IDGenerator(String prefix){ }

    // user supplies prefix
    // user supplies starting number
    public IDGenerator(String prefix, int start){ }

    // issue an id and increment number
    public String issueNext(){ }

    // return id but do not increment
    public String viewNext(){ }
}
```

Exercises

4. Write a class named `Employee,` which is composed of the following.

   - A `SimpleDate` object representing the date the employee was hired
   - A `String` object representing the name of the employee
   - A `String` representing the employee id

- The class should also have a `static IDGenerator` object that will be used by the `Employee` constructor to assign an id to each employee created.
- The starter code shown below can be found in the starters directory for this chapter.

Employee.java

```
package starters.encapsulation;
public class Employee {
    // instance data
    private String name;
    private String id;
    private SimpleDate hireDate;

    // static data
    private static IDGenerator idg
        = new IDGenerator("emp", 1);

    // constructors
    public Employee (String name, SimpleDate hd){ }

    // methods
    public String getName() { }

    public String getID() { }

    public SimpleDate getHireDate() { }

    public String toString() { }
}
```

This Page Intentionally Left Blank

Chapter 7: Inheritance & Polymorphism

Introduction

- Often, a class is needed, which is a specialization of an existing class.
  - In object-oriented languages such as Java, you can build a specialized class in such a way that the methods from the existing class can be reused without being re-coded.
  - The new class can add methods to implement its special behavior. Data can also be added to the data that is inherited from the original class.
- The building of specialized classes from existing classes in this way is known as **inheritance**.
  - The relationship between the existing class and the newly created class is called the **is-a** relationship.
- In Java, the two classes are referred to as the **subclass** and **superclass**.
  - In this terminology, the superclass is the original class.
    - Other terminology refers to these two classes as the **type** and **sub-type**.
    - C++ uses the terms **base class** and **derived class**.
- This process can be repeated giving rise to a hierarchy of classes originating from one class.
- Inheritance gives the advantage of code reuse (i.e., new classes can use existing methods from classes higher in the hierarchy).
  - Inheritance is one of the signature characteristics of object-oriented languages.

A Simple Example

- Suppose the need arises to create a class named `Point3D` to represent a three-dimensional point.
  - Since we already have a class named `Point`, we can derive a `Point3D` from `Point`.
- The `extends` keyword is used to derive one class from another.
  - The newly derived `Point3D` class (the subclass) will have additional data and methods not found in the `Point` class (the superclass).
  - Keep in mind that a `Point3D` inherits the methods and data items of the `Point` class, and is responsible for initializing all of its inherited data members.
  - A skeleton version of this class might look like the following.

Point3D.java

```
package examples.inheritance;
public class Point3D extends Point {
    private int zc;

    public Point3D(int xc, int yc, int zc){ }

    public int getZc() { }

    public String toString() { }
}
```

A Simple Example

- The super keyword is used to call a constructor from the superclass to initialize the inherited instance data.

```
public Point3D(int xc, int yc, int zc){
    super(xc, yc);
    this.zc = zc;
}
```

  - The functional form of super can only be used inside of a constructor and must be the first statement in the constructor.
  - If the super call is omitted, the compiler automatically calls the constructor in the superclass that takes no parameters.
    - If such a constructor does not exist in the superclass, the subclass will not compile.

- Since the variables xc and yc, inherited from the Point class, are defined as private, trying to define the toString method in the Point3D class as follows would result in an access level error at compile time.

```
public String toString() {
    return xc + "," + yc + "," + zc;
}
```

  - Since the getXc and getYc methods that are inherited are defined as public, the toString method could be defined as follows.

```
public String toString() {
    return getXc() + "," + getYc() + "," + zc;
}
```

  - The following form of super provides a simpler definition.

```
public String toString() {
    return super.toString() + "," + zc;
}
```

A Simple Example

- The completed `Point3D` class is shown below.

Point3D.java

```
package examples.inheritance;
public class Point3D extends Point {
    private int zc;

    public Point3D(int xc, int yc, int zc){
        super(xc, yc);
        this.zc = zc;
    }

    public int getZc() {
        return zc;
    }

    public String toString() {
        return super.toString() + "," + zc;
    }
}
```

- Notice that the `Point3D` class does not define methods `getXc` and `getYc` but instead inherits them from the `Point` class.
- This is a good example of code reuse and is one of the major benefits of inheritance.
- The application below tests the `Point3D` class defined above.

Test3D.java

```
package examples.inheritance;
public class Test3D {
    public static void main(String args[]) {
        Point3D p1 = new Point3D(1,2,3);
        System.out.println(p1);
        System.out.println(p1.getXc());
        System.out.println(p1.getYc());
        System.out.println(p1.getZc());
    }
}
```

The `Object` Class

- Java classes ultimately derive from a root class whose name is `Object`.
  - This means that even if you do not explicitly use the `extends` keyword, your class extends `Object`.
  - Therefore, the following two class definitions are equivalent.
    - Implicitly extending from the `Object` class

      ```
      public class Point {
      }
      ```

    - Explicitly extending from the `Object` class

      ```
      public class Point extends Object{
      }
      ```

- This means that all classes inherit some methods from the `Object` class.
  - One of the methods is the `toString` method.
    - If your class does not define a `toString` method, the one from the `Object` class is used.
    - The `toString` method inherited from the `Object` class is designed to print the name of the class, followed by the `@` symbol, followed by a hashcode, uniquely identifying the object.
    - Since the above representation is usually not desired, each class would typically provide its own version of the `toString` method as we have been doing.
  - Another method provided by the `Object` class is the `equals` method.
    - The method as defined in the `Object` class simply tests references, not the actual data inside of the object.

The `Object` Class

- The application below demonstrates the behavior inherited from the `equals` method in the `Object` class.

EqualityTest.java

```
package examples.inheritance;
public class EqualityTest {
    public static void main(String args[]) {

        Point p1 = new Point(2,3);
        Point p2 = new Point(2,3);
        Point p3 = new Point(7,8);

        if ( p1.equals(p2) )
            System.out.println(p1 + " = " + p2);
        else
            System.out.println(p1 + " != " + p2);

        if ( p1.equals(p3) )
            System.out.println(p1 + " = " + p3);
        else
            System.out.println(p1 + " != " + p3);

        p1 = p3;
        if ( p1.equals(p3) )
            System.out.println(p1 + " = " + p3);
        else
            System.out.println(p1 + " != " + p3);
    }
}
```

```
java examples.inheritance.EqualityTest
2,3 != 2,3
2,3 != 7,8
7,8 = 7,8
```

- Much like the `toString` method, if the behavior of the inherited `equals` method is not what you would prefer, the `Point` class would provide its own version of the `equals` method as shown on the next page.

Method Overriding

- The `Point` class might provide its own definition of the `equals` method, which checks the data in the objects rather than just the references as shown below.

```
public boolean equals (Point p) {
    return this.xc == p.xc && this.yc == p.yc;
}
```

- The fact that a subclass has its own version of a method in a superclass is called **method overriding**.
  - A method which overrides another method must have the exact signature as the overridden method.

- Do not confuse method overriding with method overloading, which we studied earlier.
  - **Method overloading**
    - occurs when two or more methods in the same class have the same name but a different parameter list.
  - **Method overriding**
    - occurs when methods from different classes in an inheritance hierarchy have the same name and the same parameter list.

- The `toString` method defined in the `Point` class uses the functionality of the `toString` method from its superclass in the process of overriding it to change its behavior.

```
public String toString() {
    return super.toString() + "," + zc;
}
```

Polymorphism

- Recall that the inheritance relationship models the is-a relationship.
  - Since a variable defined as type `Point` is capable of referencing any `Point` object, and a `Point3D` is-a `Point`, it follows that a variable of type `Point` should be able to reference a `Point3D` object.

    ```
    Point p;
    p = new Point3D(1, 2, 3);
    ```

  - However, what happens if we now try to print the object to which `p1` points? Which `toString` method gets called?
    - During the compilation of the program, `p1` is defined as a reference of type `Point`, and therefore one might argue that the `toString` method in the `Point` class will be called.
    - During the execution of the program, `p1` points to a `Point3D` object and therefore another might argue that the `toString` method in the `Point3D` class is used.
- In Java, it is always the run-time (or late) type to which the object is being pointed which dictates the correct method to use.
  - This concept is called **polymorphism** due to the fact that the reference can refer to more than one type.
- Polymorphism provides great flexibility and low maintenance to programs.
  - The example on the next page demonstrates one of the advantages of polymorphism.

Polymorphism

PolyTest.java

```
package examples.inheritance;
public class PolyTest {
    public static void main(String args[]) {
        Point data [] = { new Point(1,2),
                          new Point3D(1,2,3),
                          new Point(2,3),
                          new Point3D(2,3,4) };

        for (int i = 0; i < data.length; i++)
            System.out.println(data[i]);
    }
}
```

java PolyTest

```
1,2
1,2,3
2,3
2,3,4
```

- The important thing to notice above is that the correct `toString` method was called without the program needing to concern itself with the type of the reference.
    - If there is a need to determine the actual run-time type of the object being referenced, the `instanceof` operator can be used.

```
for (int i = 0; i < data.length; i++){
    if( data[i] instanceof Point3D)
        System.out.println("a Point3D object");
}
// Example of a cast to recognize the specialized
//behavior of the subclass
for (int i = 0; i < data.length; i++){
    if( data[i] instanceof Point3D){
        Point3D temp = (Point3D) data[i];
        System.out.println(temp.getZc());
    }
}
```

Additional Inheritance Examples

- Say that we needed to define a couple of new classes called a `CarLoan` and a `BusinessLoan.`
  - These classes could be designed from the ground up, or we could re-use the functionality already designed into the `Loan` class studied earlier.
    - Both a `CarLoan` and a `BusinessLoan` can be thought of as special types of the `Loan` class with some extra data and methods.
- The `CarLoan` can simply extend the `Loan` class as shown below.

`CarLoan.java`

```
package examples.inheritance;
public class CarLoan extends Loan {
    private String make;
    private String model;
    public CarLoan(String n, double a, double r,
                   int y, String mk, String mod) {
        super(n, a, r, y);
        make = mk;
        model = mod;
    }
    public String getMake() { return make;  }
    public String getModel(){ return model; }
    public String toString(){
        StringBuffer sb
            = new StringBuffer(super.toString());
        sb.append(", ");
        sb.append(make);
        sb.append(", ");
        sb.append(model);
        return sb.toString();
    }
}
```

Additional Inheritance Examples

- The `BusinessLoan` can also extend the `Loan` class as shown below.

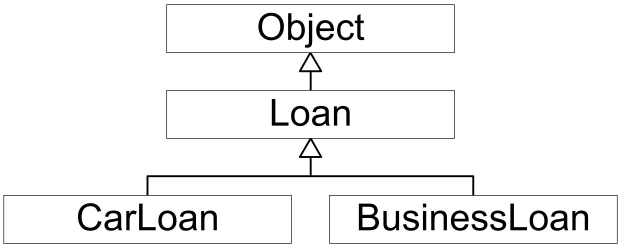
BusinessLoan.java

```java
package examples.inheritance;
public class BusinessLoan extends Loan {
    private int zipCode;
    private double sales;
    public BusinessLoan(String n, double a, double r,
                        int y, int z, double sa) {
        super(n, a, r, y);
        zipCode = z;
        sales = sa;
    }
    public int getZip() {
        return zipCode;
    }
    public double getSales() {
        return sales;
    }
    public String toString(){
        StringBuffer sb
            = new StringBuffer(super.toString());
        sb.append(", ");
        sb.append(zipCode);
        sb.append(", ");
        sb.append(sales);
        return sb.toString();
    }
}
```

- Other loan types are possible as well.
  - The new loan types could extend directly from the `Loan` class.
  - They might also extend from one of the newly designed subclasses such that a `NewCarLoan` and a `UsedCarLoan` class might extend from `CarLoan`.

Additional Inheritance Examples

- The class hierarchy for the Loan class and its subclasses is shown below.



- The application below demonstrates how any method that accepts a `Loan` as an argument will also accept a `CarLoan` or `BusinessLoan` as an argument.
  - It also demonstrates the ability to obtain the actual run-time type of the object being referenced and the steps necessary to then call methods on that object that are not inherited from the superclass.

LoanTests.java

```
package examples.inheritance;
public class LoanTests{
    public static void main(String args[]){

        Loan loans[] = {
            new Loan("Susan", 50000, 6.0, 30),
            new BusinessLoan("Alan", 75000, 7.0,
                             30, 21046, 1000000),
            new CarLoan("Michael", 30000, 4.5,
                        5, "Mazda", "Miata")};
        for(int i = 0; i < loans.length; i++){
            displayLoanInfo(loans[i]);
        }
        System.out.print("Total interest: ");
        System.out.println(calcInterest(loans));
    }

```

Additional Inheritance Examples

LoanTests.java ***continued***

```
    public static void displayLoanInfo(Loan loan){
        print("  Name : " + loan.getName());
        print("Amount : " + loan.getAmount());
        print("  Rate : " + loan.getRate());
        print("Years  : " + loan.getYears());
        print("Payment: " + loan.computePayment());
        print("");
    }

    public static double calcInterest(Loan loans []){
        double total = 0.0;
        for(int i = 0; i < loans.length; i++){
            total += loans[i].totalInterest();
        }
        return total;
    }

    public static void print(String s){
        System.out.println(s);
    }
}
```

Other Inheritance Issues

- This section has revealed the details of **inheritance of implementation**.
  - Subclasses inherit implementations from superclasses by extending them.
- Subclasses are free to use each method from its superclass as a default, or they may override any needed methods.
  - For example, each class will usually override the `toString` and the `equals` methods inherited from the `Object` class.
- Any class can extend only one other class. That is, there is no multiple inheritance of implementation in Java.
- If you wish to prevent a class from being extended, the `final` keyword can be used as demonstrated below.

```
public final class MyClass{
    // data and methods of class here
}
```

- Inheritance of implementation is not the only kind of inheritance.
  - The next chapter discusses how a class can be defined to inherit one or more interfaces.
  - This is known as multiple **inheritance of interface**.

Exercises

1. Extend the `SimpleDate` class to create a `Holiday` and `Appointment` class.

    - A `Holiday` is a `SimpleDate` with an associated `String` for the name of the holiday.
    - An `Appointment` is a `SimpleDate` with the following.
        - A `String` for the place of the appointment
        - A `String` for the name of the person with whom the appointment is made

2. Create a class named `Planner` that is composed of a single array containing any mixture of `SimpleDate`, `Holiday`, and `Appointment` objects.

    - The starter file shown below can be found in the starters directory for this chapter.

Planner.java

```
package starters.inheritance;
public class Planner{
    private SimpleDate sd [];
    private int capacity;
    private int size;

    public Planner(int capacity){ }

    public int getCapacity(){ }

    public int getSize(){ }

    public void addDate(SimpleDate d){ }

    public String toString(){ }

    public Appointment [] getAppointments(){ }

    public Holiday [] getHolidays(){ }
}
```

Chapter 8:
Abstract Classes and Interfaces

Introduction

- A set of classes in an inheritance hierarchy is not built overnight. It takes much thought and a lot of design.

- One way of thinking about a large set of classes is to factor out the functionality that would apply to all classes and let this functionality "bubble" to the top of the class hierarchy.
  - For example, a large set of classes in a `DataStructure` hierarchy might have an `isEmpty` method and a `length` method.
  - Therefore, it would make sense for this method to be encoded at the top of the hierarchy and reused by all subclasses.

- Likewise, a `calcArea` method and a `calcPerimeter` method might also seem like candidates for a top-level class in a `Shape` hierarchy.
  - However, these methods would have a different implementation in each subclass because every concrete `Shape` has it own way of calculating its area and perimeter.
  - We could still factor up the interface for the `calcArea` and `calcPerimeter` methods and leave the actual implementation to each class that extends `Shape`.

- This section describes how all of this is done in Java.

Abstract Classes

- All of the classes we have seen so far are concrete in that they represent real things.

- In contrast, some classes are more general and represent an abstraction.
  - For example, a shape is an abstract notion compared to a circle, a square, or rectangle, all of which are concrete.
  - Yet, a `Shape` class can represent the behavior of a set of concrete classes.
  - In Java, you can represent this behavior with some abstract methods (i.e., methods that have no behavior).

- Therefore, an abstract class represents an abstract concept and cannot be instantiated, but rather subclassed.
  - An abstract class may consist of one or more methods that are abstract and describe a programming interface.
  - The implementation for these abstract methods is left to classes that extend this abstract class.
  - Abstract classes can also have real methods and data.

- The following example defines an abstract class named `DataStructure`.
  - There is one `abstract` method named `addElement` whose implementation will be provided by any subclass of `DataStructure`.
  - The class also contains some concrete methods and data that are inherited by any subclass of `DataStructure`.

Abstract Class Example

DataStructure.java

```java
package examples.abstract;
public abstract class DataStructure{
    // protected data:
    // only subclasses have direct access
    protected int size = 0;

    // abstract method whose implementation
    // is left to the subclass to define
    public abstract boolean addElement(int element);

    // concrete methods that are
    // inherited by any subclass
    public boolean isEmpty(){
        return size == 0;
    }

    public int size(){
        return size;
    }
}
```

- Each class that is derived from `DataStructure` would have a mandate to define the `addElement` method.
  - The `isEmpty` and `size` methods would be inherited and reused by each class derived from `DataStructure`.

- The next page defines two classes, `IntList` and `IntSet`, which are concrete implementations of `DataStructure`.
  - `IntList` implements the `addElement` method in such a way that duplicate values can be added.
  - `IntSet` implements the `addElement` method in such a way that duplicate values are not permitted.

Extending an Abstract Class

- The example below is a concrete implementation of the `DataStructure` class.

`IntList.java`

```
package examples.abstract;
public class IntList extends DataStructure {
    private int capacity;
    private int data [];

    public IntList(int capacity){
        this.capacity = capacity;
        data = new int[capacity];
    }

    public boolean addElement(int element){
        if(size < capacity){
            data[size] = element;
            size++;
            return true;
        }
        else
            return false;
    }

    public String toString(){
        StringBuffer sb = new StringBuffer();
        for(int i = 0; i < size; i++){
            sb.append(data[i]);
            if(i < size -1)
                sb.append(", ");
        }
        return sb.toString();
    }
}
```

Extending an Abstract Class

- The example below is another concrete implementation of the `DataStructure` class.

IntSet.java

```
package examples.abstracts;
public class IntSet extends DataStructure {
    private int capacity;
    private int data [];

    public IntSet(int capacity){
        this.capacity = capacity;
        data = new int[capacity];
    }
    public boolean addElement(int element){
        boolean unique = true;
        boolean success = false;
        if(size < capacity){
            for(int i = 0; i < size; i++){
                if(data[i] == element){
                    unique = false;
                    break;
                }
            }
            if(unique){
                data[size] = element;
                size++;
                success = true;
            }
        }
        return success;
    }
    public String toString(){
        StringBuffer sb = new StringBuffer();
        for(int i = 0; i < size; i++){
            sb.append(data[i]);
            if(i < size -1)
                sb.append(", ");
        }
        return sb.toString();
    }
}
```

Extending an Abstract Class

- Since `DataStructure` is abstract, the following would result in a compiler error because you cannot create an object of this type.

```
DataStructure ds = new DataStructure();
```

- However, you are able to define a variable of type `DataStructure`, as long as it ultimately references an object of a subclass that implements the abstract methods in the `DataStructure`.

```
DataStructure myList = new IntList(100);
DataStructure mySet = new IntSet(50);
```

- The application below creates a `DataStructure` from all of the values passed in on the command line.

DataStructureTest.java

```
package examples.abstracts;
public class DataStructureTest {
    public static void main(String args[]){
        DataStructure myData =
            new IntList(args.length);
        int x;
        for(int i = 0; i < args.length; i++){
            x = Integer.parseInt(args[i]);
            myData.addElement(x);
        }
        System.out.println("Size =" + myData.size());
        System.out.println(myData);
    }
}
```

  - If duplicates are not desired, the only change needed to the above application is the type of `DataStructure` created.

```
DataStructure myData = new IntSet(args.length);
```

Interfaces

- An **interface** is a collection of abstract methods and possibly, some static constant values.
    - Interfaces are somewhat like abstract classes except that they can be implemented by a set of classes not related in the same inheritance hierarchy.
    - Although a class cannot extend more than one class, a class can implement more than one interface.
    - It is also possible for a class to extend one class, and in addition, implement one or more interfaces.
- The following examples demonstrate how an interface allows unrelated classes to exhibit some common behavior.
- To start, we will define three classes that are unrelated to one another.

Car.java

```
package examples.abstracts;
public class Car {
    private String make, model;
    public Car (String make, String model) {
        this.make = make;
        this.model = model;
    }
    public String getMake() { return make;  }
    public String getModel(){ return model; }
    public String toString(){
        return make + " " + model;
    }
}
```

Interfaces

Book.java

```java
package examples.abstracts;
public class Book {
    private String title, author;
    public Book (String title, String author) {
        this.title = title;
        this.author = author;
    }
    public String getTitle() { return title;  }
    public String getAuthor(){ return author; }
    public String toString(){
        return title + " " + author;
    }
}
```

Computer.java

```java
package examples.abstracts;
public class Computer {
    private String brand, chip;
    public Computer (String brand, String chip) {
        this.brand = brand;
        this.chip = chip;
    }
    public String getBrand() { return brand; }
    public String getChip()  { return chip;  }
    public String toString(){
        return brand + " " + chip;
    }
}
```

- Suppose we wanted to create an application capable of auctioning items to the highest bidder.
  - To list the items in an auction, it would be convenient if the application could describe each item and its condition.
    - As long as the application can obtain the description and condition of an item, the item can be auctioned.
    - The auction application can then maintain the high bidder and high bid for any object that is in the auction.

Interfaces

- We will create an interface named `Auctionable` to define the methods that are necessary for the application to interact with any item that is available to be auctioned.

`Auctionable.java`

```
package examples.abstracts;
public interface Auctionable {
    // available conditions
    public static final int NEW         = 0;
    public static final int LIKE_NEW    = 1;
    public static final int REFURBISHED = 2;
    public static final int USED        = 3;

    // abstract methods to be implemented
    public String getDescription();
    public int getCondition();
}
```

  - The static constants defined in the `Auctionable` interface set the acceptable conditions for an `Auctionable` object.

- The Auction application on the next page demonstrates how any object that implements the `Auctionable` interface can be auctioned to the highest bidder.
  - The class as written will not compile due to a problem in the `main` method shown below.

```
public static void main(String args[]){
    Auctionable a = new Car("Ford", "Mustang");
    auctionIt(a);
}
```

    - The `Car` class does not yet implement the `Auctionable` interface.

Interfaces

Auction.java

```
package examples.abstracts;
public class Auction{
    public static String [] people =
        {"Joe", "Sue", "Lynn", "Bob"};
    public static String [] conditions =
        {"New", "Like New", "Refurbished", "Used"};

    public static void main(String args[]){
        Auctionable a = new Car("Ford", "Mustang");
        auctionIt(a);
    }
    public static void auctionIt(Auctionable item){
        double highBid = 0;
        String highBidder = null;

        // bidding process
        for(int i = 0; i < people.length; i++){
            double bid = getRandomBid();
            print(people[i] + " bidding " + bid);
            if(bid > highBid){
                highBidder = people[i];
                highBid = bid;
            }
        }
        print("------------");
        print("Auction Results:");
        print("Item: " + item);
        print("Desc: " + item.getDescription());
        int c = item.getCondition();
        print("Condition: " + conditions[c]);
        print("HighBidder: " + highBidder);
        print("HighBid: " + highBid + "\n\n");
    }
    public static double getRandomBid(){
        int x = (int) (Math.random() * 10000);
        double b = x / 100.0;
        return b;
    }
    public static void print(String s){
        System.out.println(s);
    }
}
```

Interfaces

- The `Car` class is rewritten below so that it successfully implements the `Auctionable` interface.

Car.java

```
package examples.abstracts;
public class Car implements Auctionable{
    private String make, model;
    public Car (String make, String model) {
        this.make = make;
        this.model = model;
    }
    public String getMake() { return make;  }
    public String getModel(){ return model; }
    public String toString(){
        return make + " " + model;
    }
    public String getDescription() {
        return "Low Mileage, New tires, AM/FM/CD";
    }
    public int getCondition(){
        return LIKE_NEW;
    }
}
```

- The output from running the `Auction` is shown below.

```
java examples.abstracts.Auction
Joe bidding 34.4
Sue bidding 4.91
Lynn bidding 44.12
Bob bidding 42.16
------------
Auction Results:
Item: Ford Mustang
Desc: Low Mileage, Brand new tires, AM/FM/CD
Condition: Like New
HighBidder: Lynn
HighBid: 44.12
```

  - Keep in mind, `Auction.java` could have been written to handle an array of type `Auctionable`.

Exercises

1. Define a class named `SortedIntList` that `extends` the abstract class `DataStructure` from this chapter.
   - The addElement method should be implemented in such a way that the `int` values of the array are maintained in sorted order.
   - The existing `DataStructureTest` can be modified to test your `SortedIntList`.
2. Modify the `Book` and `Computer` classes from this chapter so that they can be auctioned off using the `Auction` application.
3. Modify the `Auction` application so that it deals with an array of type `Auctionable` rather than a single `Auctionable` object.
   - Test this class with several `Car`, `Book`, and `Computer` objects.

This Page Intentionally Left Blank

Chapter 9: Exceptions

Introduction

- In this section, we wish to explore how a Java Program reacts to runtime errors.
  - Some examples of the causes of a runtime error are:
    - data in the wrong format;
    - file open failure;
    - bad network connection;
    - division by zero.
  - Traditional error handling is usually done in a disorderly fashion.
    - Error code is often spread out in a program.
    - Often, there are too many branches of control.
    - Normal program flow is buried within error detection code.
  - In Java, error-handling transfers program control to an error handling routine in an orderly fashion and to a well-defined section of code.
    - Java syntax for handling run-time errors looks very much like C++ syntax.
- The Java Exception model is built around the following keywords.
  - `try`
  - `catch`
  - `throw`
  - `throws`
  - `finally`

Introduction

- The `try` statement
  - It identifies a block of statements within which an exception might be thrown.
- The `catch` statement
  - It identifies a block of statements that can handle a particular type of exception.
  - The statements are executed if an exception of a particular type occurs within the `try` block.
  - If `catch` statement is used, it must be associated with a `try` statement.
- The `finally` statement
  - It identifies a block of statements that are executed regardless of whether or not an error occurs within the `try` block.
  - If `finally` statement is used, it must be associated with a `try` statement.

- The general form of these statements is shown below.

```
try {
    // execute a method which may throw an exception
} catch (Type1Exception e) {
    // code to handle a Type1Exception
} catch (Type2Exception e) {
    // code to handle a Type2Exception
} finally {
    // code to be executed regardless of whether an
    // exception occurred or not
}
```

- Each `catch` block is an exception handler. Therefore, for each `try` block, there can be as many `catch` blocks as there are different exception types handled.

Exception Handling

- The following program takes two arguments from the command line and raises the first number to the power of the second.
  - Several things could go wrong at runtime.
    - The user may not supply the correct number of arguments.
    - The data supplied might not be numeric.

Raise.java

```
 1.  package examples.exceptions;
 2.  public class Raise {
 3.      public static void main(String args[]) {
 4.          double base, expo, result;
 5.
 6.          base = Double.parseDouble(args[0]);
 7.          expo = Double.parseDouble(args[1]);
 8.          result = Math.pow(base, expo);
 9.          System.out.println(result);
10.      }
11.  }
```

  - The output from running the above program with various arguments is shown below.

```
java examples.exceptions.Raise 5
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 1
        at examples.exceptions.Raise.main(Raise.java:7)

java examples.exceptions.Raise one two
Exception in thread "main" java.lang.NumberFormatException:
For input string: "one"
  at java.lang.NumberFormatException.forInputString(
    NumberFormatException.java:48)
  at java.lang.FloatingDecimal.readJavaFormatString(
    FloatingDecimal.java:1207)
  at java.lang.Double.parseDouble(Double.java:220)
  at examples.exceptions.Raise.main(Raise.java:6)
```

  - In both cases, the JVM handles the runtime exception.

Exception Handling

- The following code demonstrates how to handle the exceptions from the `Raise` class.

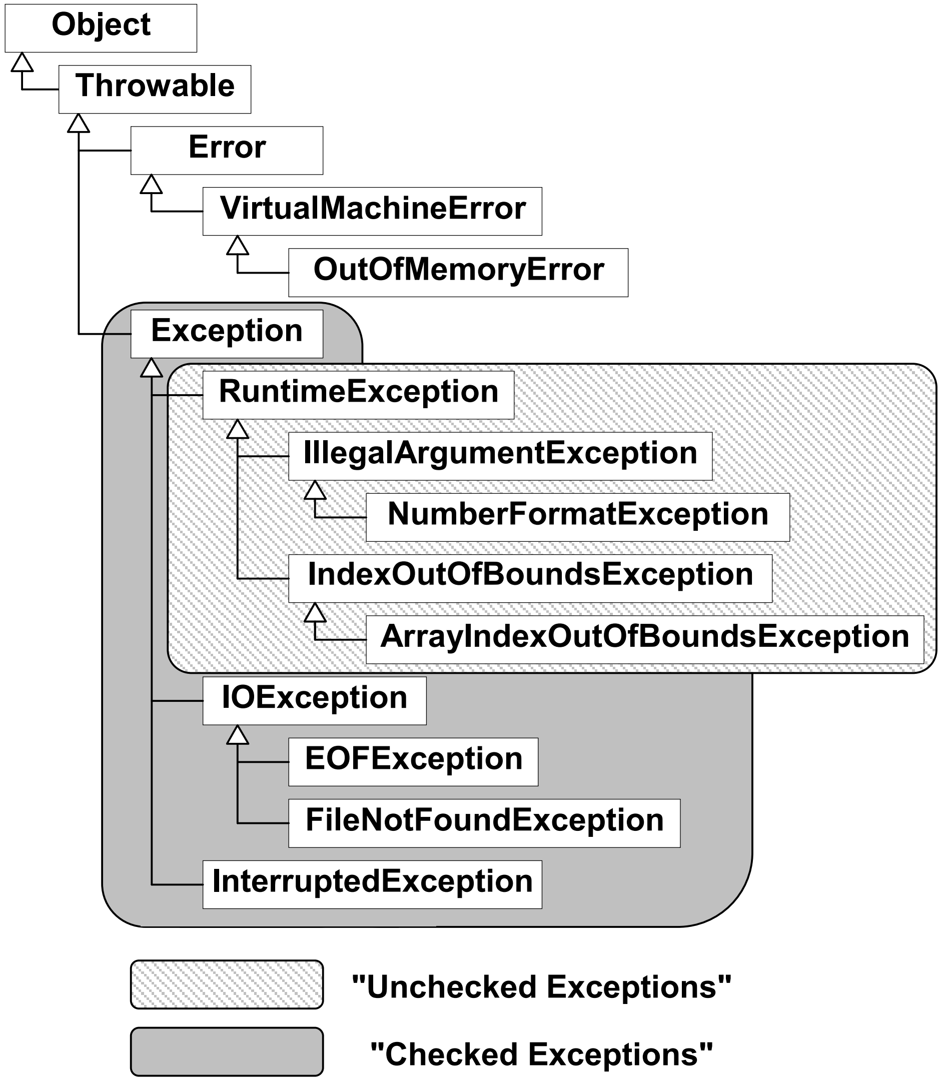
RaiseWithHandler.java

```
package examples.exceptions;
public class RaiseWithHandler {
    public static void main(String args[]) {
        double base, expo, result;

        try {
            base = Double.parseDouble(args[0]);
            expo = Double.parseDouble(args[1]);
            result = Math.pow(base, expo);
            System.out.println(result);
        } catch(NumberFormatException nfe) {
            System.out.println(nfe);
        } catch(ArrayIndexOutOfBoundsException ai) {
            System.out.println(ai);
        } finally {
            System.out.println("Inside of finally");
        }
        System.out.println("Finished with handlers");
    }
}
```

  - Keep in mind that should an exception be raised, you will always have the choice of either letting the JVM handle it or handling it yourself.
  - When you wish to write your own handlers, you need to decide which code to enclose in a `try` block.
    - This triggers the exception handling mechanism.
  - `try` blocks must be followed by any of the following.
    - one or more `catch` blocks
    - a `finally` block
    - one or more `catch` blocks followed by a `finally` block

The Exception Hierarchy

- A subset of the hierarchy of the exception classes is shown below.



- The superclass `Throwable` provides many of the methods commonly used in `catch` blocks.
  - `getMessage()` returns the message associated with the exception.
  - `printStackTrace()` prints the origin of the exception.
  - `toString()` returns the exception name and message.

Checked Exceptions

- The JVM throws an exception in the following cases.
  - An internal error has occurred.
  - You make a programming error, such as an out-of-bounds error.
  - You detect an error and execute a `throw.`
  - You call a method that `throws` an exception.
- `Error` and its subclasses typically handle exceptions thrown by the JVM.
  - These are errors from which you do not intend to recover.
- `RuntimeException` and its subclasses are called **unchecked exceptions**.
  - The Java compiler does not need to know how you plan on handling these exceptions should they arise.
- `Exception` and all of its subclasses (excluding `RuntimeException` and its subclasses) are called **checked exceptions**.
  - The Java compile does need to know how you plan on handling checked exceptions. If you do not specify how you plan on handling these exceptions, the compiler will issue an error message.
  - A programmer can either:
    - handle them; or
    - send them up the call stack.

Checked Exceptions

- Therefore, it is necessary to know for which methods you are using throw checked exceptions.
  - If you do not know, the compiler will inform you at compile time of any unreported checked exceptions as demonstrated in the following example.
  - The `Thread` class has a `sleep` method which allows you to sleep for a specific number of milliseconds.

PrintHello.java

```
1.  package examples.exceptions;
2.  public class PrintHello {
3.      public static void main(String args[]) {
4.          while(true) {
5.              System.out.println("hello");
6.              Thread.sleep(2000);
7.          }
8.      }
9.  }
```

- When this program is compiled, the compiler will indicate the following error.

```
PrintHello.java:5: unreported exception
java.lang.InterruptedException; must be caught or
declared to be thrown
            Thread.sleep(2000);
                        ^
```

- If you look in the API docs, the `sleep` method in the `Thread` class is defined as:

```
public static void sleep(long millis)
    throws InterruptedException
```

  - Since `InterruptedException` extends from `Exception`, it is a checked exception.
  - The examples on the next page demonstrate the two ways the exception can be handled.

Checked Exceptions

- Below is an example of passing a checked exception up the call stack.

PrintHello2.java

```
package examples.exceptions;
// Passing it up the call stack
public class PrintHello2 {
    public static void main(String args[])
        throws InterruptedException{
        while(true) {
            System.out.println("hello");
            Thread.sleep(2000);
        }
    }
}
```

- An example of handling a checked exception with an exception handler is shown below.

PrintHello3.java

```
package examples.exceptions;
// Handling it with an exception handler
public class PrintHello3 {
    public static void main(String args[]) {
        while(true) {
            System.out.println("hello");
            try {
                Thread.sleep(2000);
            } catch(InterruptedException ie){
                ie.printStackTrace();
            }
        }
    }
}
```

Advertising Exceptions with `throws`

- The example below demonstrates the case where a method calls another method that can throw a checked exception.

`CallStack.java`

```
package examples.exceptions;
public class CallStack {
    public static void main(String args[])
      throws InterruptedException {
        methodA();
    }

    public static void methodA()
      throws InterruptedException {
        System.out.println("hello");
        Thread.sleep(2000);
    }
}
```

- In the above code, the `methodA` method calls the `sleep` method.
  - Since the `sleep` method throws a checked exception, `methodA` has to indicate whether it is handling it or passing it up the call stack.
  - In the above code, `methodA` is passing it up the call stack to whichever method calls it.

- Since the `main` method calls `methodA`, the `main` method now has to indicate how it intends to deal with the exception thrown by `methodA`.
  - The `main` method is passing it up the call stack to whichever method calls it.

- Since the JVM that calls the `main` method, the JVM ultimately handles the `InterruptedException`.

Developing Your Own Exception Classes

- If none of the existing `Exception` subclasses convey the necessary information about a particular error situation, you can derive your own subclass.
  - If you want your exception to be an unchecked exception, your class should extend from `RuntimeException` or one of its subclasses.
    - These exceptions are used when there is no reasonable possibility that a program can recover from the exception.
  - If you want your exception to be a checked exception, your class should extend from `Exception` or one of its subclasses (other than `RuntimeException`).
    - These exceptions are used for those cases when the programmer could have written code to handle these errors.
- Recall the `setRadius` method in the `Circle` class that was designed earlier.

```
public void setRadius(int r){
    if(r < 0){
        print("Bad radius: " + r);
        print("radius " + radius + "unchanged" );
        return;
    }
    radius = r;
}
```

  - As written, it is hard to distinguish between the error detection and the actual setting of the radius.

Developing Your Own Exception Classes

- A better design for the `setRadius` method may be to throw an `Exception`, should a negative value be passed as the parameter.
  - The example below defines a new data type to encapsulate the concept of a negative radius being an exception.

NegativeException.java

```
package examples.exceptions;
public class NegativeException extends Exception {
      // instance variable to hold the negative
      // value that represents the error
      private int n;

      // Constructor
      public NegativeException(String msg, int num) {
        // parent class already knows how to handle
        // the message so we will pass it to the
        // constructor in our parent class
        super(msg);

        // we will handle the number here
        n = num;
      }
      public int getNegativeValue() {
        return n;
      }
}
```

- Now, the `setRadius` method of the `Circle` class can be simplified as shown below.

```
public void setRadius(int r)
  throws NegativeException {
    if(r < 0)
        throw new NegativeException("Bad Radius", r);
    radius = r;
}
```

  - The new version of the `Circle` class that incorporates the `NegativeException` is shown on the next page.

Developing Your Own Exception Classes

Circle.java

```java
package examples.exceptions;
public class Circle {
    private int xc, yc, radius;
    public Circle(int x, int y, int rad)
      throws NegativeException {
        xc = x;
        yc = y;
        if(rad < 0)
            throw new NegativeException("Bad Radius",
                                         rad);
        radius = rad;
    }
    public Circle(int x, int y)
      throws NegativeException {
        this(x, y, 1);
    }
    public Circle(int rad)
      throws NegativeException {
        this(0, 0, rad);
    }
    public Circle()
      throws NegativeException {
        this(0, 0, 1);
    }
    public double calcArea() {
        return Math.PI * radius * radius;
    }
    public String toString() {
        return xc + "," + yc + ": rad = " + radius;
    }
    public int getXc()    { return xc; }
    public int getYc()    { return yc; }
    public int getRadius(){ return radius; }

    public void setRadius(int r)
      throws NegativeException {
        if(r < 0)
            throw new NegativeException("Bad Radius",
                                         r);
        radius = r;
    }
}
```

Developing Your Own Exception Classes

- The two programs below demonstrate handling the `NegativeException` and passing it up the call stack.

`TestCircle1.java`

```
package examples.exceptions;
public class TestCircle1 {
    public static void main(String args[]){
        try{
            Circle c = new Circle(1, 1, -5);
        } catch (NegativeException ne) {
            int x = ne.getNegativeValue();
            System.out.println(x);
            System.out.println(ne);
        }
    }
}
```

`TestCircle2.java`

```
package examples.exceptions;
public class TestCircle2 {
    public static void main(String args[])
      throws NegativeException {
        Circle c = new Circle(1, 1, -5);
    }
}
```

The `finally` Block

- The last step in setting up an exception handler is providing a mechanism for cleaning up the state of the method before (possibly) allowing control to be passed to a different part of the program.
  - This is done by enclosing the cleanup code within a `finally` block.
  - Code inside of a `finally` block is executed whether the exception is thrown or not.
    - The only time code inside of a `finally` block will not be executed is when the `System.exit` method is invoked.

Exercises

1. Write a program that reads three command line arguments.

   - Your program should send the three arguments to a method, which in turn, sends them to another method.
     - The main method should catch any `ArrayIndexOutOfBoundsException`.
     - The middle method should catch any `StringIndexOutOfBoundsException`.
     - The innermost method should catch any `NumberFormatException`.
   - Though the exceptions will be caught in different places, the code should be written such that all of them are generated inside of the innermost method.

2. Define a class named `BadMonthException` that can be thrown by methods in the `SimpleDate` class if a month is less than `1` or greater than `12`.

   - `BadMonthException` should extend from `RuntimeException.`

3. Modify the `BadMonthException` class so that it is a checked exception.

   - Make all necessary changes to the methods of the `SimpleDate` that throw a `BadMonthException`.

Chapter 10:
Input and Output in Java

Introduction

- The `java.io` package contains many classes, which correspond to various ways of performing input and output in Java.

- Although there are over 50 classes in this package, most of them descend from one the root classes listed.
  - `InputStream`
  - `OutputStream`
  - `Reader`
  - `Writer`

- Subclasses of `InputStream` and `OutputStream` are referred to as **byte streams**, which are used to perform input and output of 8-bit bytes.

- Subclasses of `Reader` and `Writer` are referred to as **character streams**, which are used to perform input and output of characters, automatically handling translation to and from the local character set.

- Before we study the classes that allow actual file I/O, we will cover the first class in the `java.io` package, the `File` class.
  - The `File` class contains a `String,` representing the name of a file or directory and methods for querying information about a file.
  - The program on the next page shows some of the methods from the `File` class.

The **`File`** Class

FileStatus.java

```
package examples.io;
import java.io.*;
import java.util.*;
public class FileStatus  {
    public static void main(String args[]) {
        File theFile;
        for( int i = 0; i < args.length; i++) {
            theFile = new File(args[i]);
            if ( theFile.exists() )
                processFile(theFile);
            else {
                print(theFile + "Not a file");
            }
        }
    }
    public static void processFile(File f){
        print("--------------------");
        print(f + " Exists");
        print("Size: " + f.length());
        print("Path: " + f.getAbsolutePath());
        Date d = new Date(f.lastModified());
        print("Last Modified: " + d);
        if ( f.isDirectory() ){
            print("File is a directory");
            print("Files in directory are:");
            String contents [] = f.list();
            for(int i = 0; i < contents.length; i++)
                print("     " + contents[i]);
        }
    }
    public static void print(String s){
        System.out.println(s);
    }
}
```

Standard Streams

- The System class defines three standard streams.
  - `public static final InputStream in;`
  - `public static final PrintStream out;`
  - `public static final PrintStream err;`

- You might expect the standard streams to be character streams but, for historical reasons, they are byte streams.

- `System.out` and `System.err` are objects of the `PrintStream` class, which provides the `print` and `println` methods we have used throughout the course.
  - Since `PrintStream` is a subclass of `OutputStream`, it is technically a byte stream. However, `PrintStream` utilizes an internal character stream object to emulate many of the features of character streams.

- `System.in` is an object of type `InputStream`, which has no character stream features and limited functionality.
  - The `read()` method returns either the:
    - next byte of data (as an `int`) from the stream each time it is called; or
    - value `-1` indicating the end of the stream has been reached.
  - The `available()` method returns the number of bytes available from the stream.
  - The `close()` method closes the stream and releases any system resources associated with the stream.

Keyboard Input

- Below is a simple program that receives input from the keyboard (`System.in`) using the `read` method.

Keyboard.java

```
package examples.io;
import java.io.*;
public class Keyboard {
    public static void main(String args[]) {
        int val;
        try {
            while((val =  System.in.read()) != -1)
                System.out.print((char)val);
        } catch(IOException e) {
            System.err.println("Error: " + e);
        }
    }
}
```

- There are a few items to note in the above code.
  - The `read` method can throw an `IOException`. Since this is a checked exception, we enclose the method call in a try block and catch the `IOException`.
  - The `read` method returns an `int`, so we cast it to a `char` in order to display it on the screen as a character rather than a number.
  - The actual number of bytes read when typing in the following line will vary by operating system.

    `ABC` (followed by the Enter key)

    - *Windows* - Enter is a carriage return and a newline; total bytes read = 5
    - *Unix/Linux* - Enter is a newline; total bytes read = 4
    - *Macintosh* - Enter is a carriage return; total bytes read = 4

File I/O Using Byte Streams

- The next set of classes we will look at are the `FileInputStream` and the `FileOutputStream` classes. Like their superclasses (`InputStream` and `OutputStream`), these are byte streams with limited functionality.

- Use a `FileInputStream` object to open a file for reading bytes.
  - The constructors can take either a `File` or a `String` object as its parameter.

```
FileInputStream (File file)
FileInputStream (String name)
```

- Use a `FileOutputStream` object to open a file for writing bytes.
  - The constructors can take either a `File` or a `String` object as its parameter.
  - There are also constructors that take a `boolean` value, specifying whether you wish to append to (`true`) or overwrite (`false`) the file if it already exists.

```
FileOutputStream (File file)
FileOutputStream (String name)
FileOutputStream (File file, boolean append)
FileOutputStream (String name, boolean append)
```

- The example on the next page shows how the `FileInputStream` and `FileOutputStream` can be used to copy a file byte-by-byte.

File I/O Using Byte Streams

FileCopy.java

```
package examples.io;
import java.io.*;

public class FileCopy {
    public static void main(String a[]) {
        int aByte;

        FileInputStream fis = null;
        FileOutputStream fos = null;
        try {
            fis = new FileInputStream(a[0]);
            fos = new FileOutputStream(a[1]);
            while((aByte =  fis.read()) != -1)
                fos.write(aByte);
        } catch(FileNotFoundException e) {
            System.err.println("File Not Found");
            e.printStackTrace();
        } catch(IOException e) {
            System.err.println("IOError: ");
            e.printStackTrace();
        } finally {
            try {
                if (fis != null)
                    fis.close();
                if (fos != null)
                    fos.close();
            } catch(IOException e){
                // ignore the exception
            }
        }
    }
}
```

- It is important to note that the compiler does not know whether any code inside of a try block will be successfully executed. Therefore, in the code above, the variables `fis` and `fos` are initialized to `null` so that any code accessing these two variables outside of the `try` block does not result in the compiler stating that the variables might not have been initialized.

Character Streams

- In Java, character data is handled by subclasses of `Reader` and `Writer`.

- In most cases, the subclasses of `Reader` and `Writer` have the same or similar methods as the corresponding subclasses of `InputStream` and `OutputStream`.

- The subclasses `FileReader` and `FileWriter` are preferred for text files (rather than `FileInputStream` and `FileOutputStream`), because they support 16-bit Unicode characters.

- Like their byte stream counterparts, `FileReader` and `FileWriter` offer limited functionality. For example, there is no capability to read a line of input.

- The example on the next page shows how to copy a file character-by-character. Note the similarity to the `FileCopy` program studied earlier.

File I/O Using Character Streams

TextFileCopy.java

```
package examples.io;
import java.io.*;

public class TextFileCopy {
    public static void main(String a[]) {
        int aChar;

        FileReader fr = null;
        FileWriter fw = null;
        try {
            fr = new FileReader(a[0]);
            fw = new FileWriter(a[1]);
            while((aChar =  fr.read()) != -1)
                fw.write(aChar);
        } catch(FileNotFoundException e) {
                System.err.println("File Not Found");
                e.printStackTrace();
        } catch(IOException e) {
                System.err.println("IOError: ");
                e.printStackTrace();
        } finally {
            try {
                if (fr != null)
                    fr.close();
                if (fw != null)
                    fw.close();
            } catch(IOException e){
                // ignore the exception
            }
        }
    }
}
```

Buffered Streams

- The examples we have seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying operating system. This can make a program inefficient, since such requests may involve disk access or network activity.

- To reduce this kind of overhead, the `java.io` package includes classes that implement buffered I/O streams.

- The `BufferedReader` class defines a `readLine` method that returns a `String.`
  - The `readLine` method returns `null` at the end of a file rather than a `-1.`

- To use enhanced functionality such as buffered I/O, we use a technique that allows two or more classes to work together. This technique is sometimes referred to as "wrapping" one stream with another stream.

- In the next example, we will construct a `FileReader` object, and then "wrap" it with a `BufferedReader` object so that we can read a file line-by-line.
  - Note that the constructor for `BufferedReader` can take any type of `Reader` as its parameter.

File I/O Using a Buffered Stream

BufferedFileCopy.java

```
package examples.io;
import java.io.*;
public class BufferedFileCopy {
    public static void main(String a[]) {
        String theLine;
        FileReader fr = null;
        BufferedReader br = null;
        try {
            fr = new FileReader(a[0]);
            br = new BufferedReader(fr);
            while((theLine = br.readLine()) != null){
                System.out.println(theLine);
            }
        } catch(FileNotFoundException e) {
                System.err.println("File Not Found");
                e.printStackTrace();
        } catch(IOException e) {
                System.err.println("IOError: ");
                e.printStackTrace();
        } finally {
            try {
                if (br != null)
                    br.close();
            } catch(IOException e){
                // ignore the exception
            }
        }
    }
}
```

- Note that it is only necessary to close the outermost stream (i.e., the last one to be constructed).

Keyboard Input Using a Buffered Stream

- Now, suppose you want to read a line of input from the keyboard.
    - Recall that `System.in` is an object of type `InputStream`.
    - Java provides a class named `InputStreamReader` that converts an `InputStream` to a `Reader`.

`KeyboardReadLines.java`

```
package examples.io;
import java.io.*;
public class KeyboardReadLines {
    public static void main(String a[]) {
        InputStreamReader isr = null;
        BufferedReader br = null;

        String line;
        try {
            isr = new InputStreamReader(System.in);
            br = new BufferedReader(isr);

            while(true) {
                System.out.print("Enter a line: ");
                line = br.readLine();
                if (line.equalsIgnoreCase("QUIT"))
                    break;
                System.out.println("You entered: "
                    + line);
            }
        } catch(IOException e) {
            System.out.println(e.getMessage());
        } finally {
            try {
                if (br != null)
                    br.close();
            } catch(IOException e) {
            }
        }
    }
}
```

Writing Text Files

- Recall that we have been using the PrintStream class since the beginning of the course.

  ```
  System.out.println("Hello World!");
  ```

  - The data type of the static variable out in the System class is PrintStream.
  - The print and println methods of the PrintStream class are overloaded.
    - There are print and println methods that take, as a parameter, each of the Java primitives.
    - There are also print and println methods that take, as a parameter, a String.
    - Additionally, there are print and println methods that take a parameter of type Object, which results in a call to the toString method on the object being referenced at runtime (polymorphism) .

- You can "wrap" a PrintStream around an underlying OutputStream. For example:

  ```
  FileOutputStream fos = new FileOutputStream("file.txt");
  PrintStream ps = new PrintStream(fos);
  ```

  - The style below is sometimes preferred to the code above.

    ```
    PrintStream ps =
        new PrintStream(new FileOutputStream("file.txt"));
    ```

- There is a corresponding class called PrintWriter (also containing overloaded print and println methods) that can be wrapped around an underlying Writer. The next example uses a FileWriter and a PrintWriter to save data in a text file.

Writing Text Files

WriteTextFile.java

```
package examples.io;
import java.io.*;
public class WriteTextFile {
    public static void main(String a[]) {
        int iValue = 10;
        double dValue = 12.3;

        PrintWriter pw = null;
        try {
            pw = new PrintWriter(new
                 FileWriter(a[0]));

            pw.println("The integer is " + iValue);
            pw.println("The double is " + dValue);

        } catch(FileNotFoundException e) {
            System.err.println("Can't open " + a[0]);
            e.printStackTrace();
        } catch(IOException e) {
            System.err.println("IOError: ");
            e.printStackTrace();
        } finally {
            if (pw != null)
                pw.close();
        }
    }
}
```

Exercises

1. Write a program that displays the number of characters, words, and lines in a file named on the command line.

2. Write a program that prompts the user for the name of a file. If the file does not exist or is a directory, print an appropriate message;. Otherwise, ask the user if the file should be deleted. Delete the file if the user confirms.

3. Enhance your solution to the previous exercise so that the user has two additional options.

   - Rename the file (prompt the user for the new name).

   - Create a backup copy of the file (prompt the user for the name of the backup copy).

4. Write a program that receives two file names as command line arguments. Each file contains a list of words, with one or more words on each line. The program should display only the words that are common to both files.

Exercises

5. Write a program, which has a static method that can read from any type of `InputStream`.

   - The static method should take two parameters.
     - The first parameter should be the `InputStream,` from which to read.
     - The second parameter should be an `int,` indicating how many characters to print per line before wrapping the output to a new line.
     - This static method should be called by the `main` method.
   - The `main` method will determine which parameters to pass to the static method based on how many arguments are supplied on the command line.
     - If only one argument is supplied on the command line, the input should be from the keyboard, and the wrap length should be obtained from the argument on the command line.
     - If two arguments are supplied on the command line, the first should be the wrap length and the second the name of the file from which to read.

Chapter 11: Collections

Introduction

- In this chapter, we explore the various interfaces, abstract classes, and concrete classes that make up the **Collections Framework**.

- A collection is a group of data elements managed by a single object, with operations provided to manipulate the data inside of the collection.

- Prior to JDK1.2, there existed a limited number of classes to represent and manipulate data structures in the `java.util` package.
  - `Vector` supports the concept of a growable array.
  - `Hashtable` supports the concept of an associative array.
  - `Enumeration` provides a way of getting each element in a `Vector` or `Hashtable`.

- As of JDK1.2, a new framework for collections was defined and implemented.
  - This framework standardized the architecture for representing and manipulating collections of data.
  - The Collections Framework offers the following benefits.
    - Reduced programming effort
    - Easier to pass collections between unrelated APIs
    - Increased program speed.

- Since the older classes are still in wide use, we will explore them before we study the newer collections framework.

Vectors

- The `Vector` class supports the concept of a growable array.
  - Like an array, its components can be accessed by index.
  - The size of a `Vector` can grow or shrink as needed.
  - Any `Object` can be added, removed, or returned from a `Vector.`
    - If a primitive is added, it needs to be wrapped up inside of an `Object` in order to be added to the `Vector`.
  - The example below shows some of the `Vector` class methods.

VectorTest.java

```java
package examples.collections;
import java.util.Vector;
public class VectorTest {
    public static void main(String args[])  {
        Vector v = new Vector(100);
        print("SIZE = " + v.size());
        print("CAPACITY = " + v.capacity());
        Integer x = new Integer(10);
        v.add(x);
        v.add(new Double(10.5));
        v.add("Mike");

        print("SIZE = " + v.size());
        for (int i = 0; i < v.size(); i++)
            print(v.get(i));

        print(x + " at pos: " + v.indexOf(x));
        print(v);
        v.remove(1);
        print(v);
    }
    public static void print(Object o){
        System.out.println(o);
    }
}
```

Hashtables

- The Hashtable class is used to create associated pairs.
  - Each pair has a **key** and a **value**.
    - The key can be used as an index to the value.
    - The key must be unique among all keys in the `Hashtable`.
  - Entries are placed into a `Hashtable` with the `put` method.

```
public Object put(Object key, Object value);
```

    - The first argument is the key, and the second argument is the value associated with the key.
    - The first time the `put` method is called for a particular key, the value `null` is returned.
    - For each use beyond that for the same key, the old value associated with that key is returned.
  - Later, you can use the `get` method to extract the value for a particular key.

```
public Object get(Object key);
```

    - The `get` method returns the value associated with the key as an `Object`.
    - Often, the object must be cast into the subclass type in order to call methods specific to the subtype.
  - The `remove` method can remove an element from a `Hashtable`.

```
public Object remove(Object key);
```

    - The `remove` method returns the removed value associated with the key, or `null` if the key is not in the `Hashtable`.

Hashtables

- The program below creates a `Hashtable` with several capital cities as keys and their state as values.
    - The program then reads the keys supplied on the command line and gets the associated values.

HashTest.java

```
package examples.collections;
import java.util.Hashtable;
public class HashTest  {
    public static void main(String args[]) {
        Hashtable caps = new Hashtable();
        caps.put("Providence", "RI");
        caps.put("Boston", "MA");
        caps.put("Hartford", "CT");
        for ( int i = 0; i < args.length; i++) {
            Object val  = caps.get(args[i]);
            if ( val == null)
                System.out.println(args[i] +
                        ": is not a capital");
            else
                System.out.println(args[i] +
                        " is capital of " + val);
        }
    }
}
```

- The results of running the above program are shown below.

```
java examples.collections.HashTest Providence
Annapolis Hartford
Providence is capital of RI
Annapolis: is not a capital
Hartford is capital of CT
```

Enumerations

- An `Enumeration` is an interface with two methods.
  - `public boolean hasMoreElements();`
    - This returns a `boolean,` indicating whether there are more elements in the `Enumeration.`
  - `public Object nextElement();`
    - This returns the next element of the `Enumeration.`
- Both the `Vector` and `Hashtable` classes have an `elements` method that returns an `Enumeration`.
  - The `elements` method in the `Vector` class returns an `Enumeration` of all of the elements in the `Vector`.
  - The `elements` method in the `Hashtable` class returns an `Enumeration` of all of the values in the `Hashtable`.
    - `Hashtable` also has a `keys` method that returns an `Enumeration` of all of the keys in the `Hashtable`.
- The example that follows demonstrates the use an `Enumeration` to loop through a `Hashtable` of `Account` objects.
  - The `Account` class we are using in the example is the same one used in an earlier chapter.
    - Each `Account` object has a unique account number that will be used as the key in the `Hashtable`.
    - Since the account number is a primitive `int`, we need to wrap it up inside of an `Integer` object in order to use it as the key.

Enumerations

- Below is a copy of the Account class studied earlier.

Account.java

```
package examples.collections;
public class Account {
    private static int counter = 1000;
    private String name;
    private int accountNum;

    public Account(String n) {
        name = n;
        accountNum  = counter++;
    }
    public String getName(){
        return name;
    }
    public int getAccNumber(){
        return accountNum;
    }
    public String toString() {
        return name + " has account # " + accountNum;
    }
    public static int nextNumber() {
        return counter;
    }
}
```

- For each Account object created, the accountNum will need to be stored in an Integer object to be able to use it as the key.

```
Hashtable accounts = new Hashtable();
Account val = new Account("Susan");
Integer key = new Integer(val.getAccNumber());
accounts.put(key, val);
```

- The complete code, including an Enumeration to obtain all of the Account objects in the Hashtable, is shown on the next page.

Enumerations

AccountTest.java

```
package examples.collections;
import java.util.*;
public class AccountTest {
    public static void main(String args[]) {
        Hashtable accounts = new Hashtable();
        // Obtain account names from command line
        Account val;
        Integer key;
        for(int i = 0; i < args.length; i++){
            val = new Account(args[i]);
            key = new Integer(val.getAccNumber());
            accounts.put(key, val);
        }
        Enumeration e = accounts.keys();
        while(e.hasMoreElements()){
            key = (Integer) e.nextElement();
            val = (Account) accounts.get(key);
            System.out.print(val.getName());
            System.out.print(" account number ");
            System.out.println(key.intValue());
        }
    }
}
```

Properties

- The java.util package contains a Properties class that allows the manipulation of a set of keys and their associated values called properties.
  - Although Properties extends Hashtable, the keys and values are only allowed to be of type String.

    ```
    public Object setProperty(String key, String value);
    ```

    - The setProperty method should be used in place of the inherited put method when populating the Properties object.
    - If the put method is used, and a type other than String is used for either the key or the value, the store and load methods of the Properties class will fail.
  - The store method allows the properties to be persisted to an OutputStream for later retrieval from the load method.
- The JVM maintains a Properties object that contains information about the environment in which the JVM is running.
  - The static getProperties method in the System class can be used to obtain the system Properties object.
- The two examples that follow create a Properties object with several key/value pairs and rely on the store and load methods to persist and retrieve the properties.

Properties

StoreProperties.java

```
package examples.collections;
import java.util.*;
import java.io.*;
public class StoreProperties{
  public static void main(String args[])
   throws IOException{
    FileOutputStream fos =
      new FileOutputStream(args[0]);
    Properties p = new Properties();
    p.setProperty("fontsize", "12");
    p.setProperty("fontcolor", "green");
    p.store(fos, "header comment");
    fos.close();
  }
}
```

LoadProperties.java

```
package examples.collections;
import java.util.*;
import java.io.*;
public class LoadProperties{
  public static void main(String args[])
   throws IOException{
    FileInputStream fis =
      new FileInputStream(args[0]);
    Properties p = new Properties();
    p.load(fis);
    fis.close();
    Enumeration e = p.propertyNames();
    String key, val;
    while(e.hasMoreElements()){
      key = (String) e.nextElement();
      val = (String) p.get(key);
      System.out.println(key + " " + val);
    }
  }
}
```

Collection Framework Hierarchy

- Now that you have seen some of the older data structure classes, such as `Vector` and `Hashtable`, we can look at the newer data structure classes provided as part of the Collections Framework.

- These classes are organized as a hierarchy of classes, the top-most of which are interfaces and abstract classes.
  - Most of the classes we will be studying implement either the `Collection` interface or `Map` interface.
  - We will first look at the `Collection` interface and several of its implementations, followed later by the `Map` interface.

- `Collection` is a top-level interface that defines the methods all collections must have.
  - Below are listed some of the methods in the `Collection` interface.

```
public boolean add(Object o);
public boolean addAll(Collection c);
public void clear();
public boolean contains(Object o);
public boolean containsAll(Collection c);
public boolean equals(Object o);
public boolean isEmpty();
public Iterator iterator();
public boolean remove(Object o);
public int size();
```

- `List` is a sub-interface of `Collection`, in that it extends the `Collection` interface by adding additional methods to retrieve an element based on an index.
  - An example of this is the additional add method defined below.

```
public boolean add(int index, Object o);
```

Collection Framework Hierarchy

- `AbstractCollection` is an abstract class that implements some of the functionality of `Collection`.
  - All of the methods from the Collection class, except for the `size` and `iterator` methods, are defined concretely.
    - It is up to the subclasses of `AbstractCollection` to implement the `size` and `iterator` methods.
- `AbstractList` extends `AbstractCollection` and implements some of the functionality of the `List` interface.
- Each of the abstract classes and interfaces we have discussed allow developers to define their own data structure if necessary, based upon the framework supplied, or use a concrete implementation supplied as part of the JDK.
  - `ArrayList` is a concrete implementation of `AbstractList`.
  - `LinkedList` is another concrete implementation that extends from `AbstractSequentialList`.
    - These two concrete implementations differ with respect to the way in which elements are stored and retrieved.
    - The `List` interface provides a description of what common functionality is provided. `ArrayList` and `LinkedList` implement the functionality in different ways.
- The example on the following page demonstrates using an `ArrayList` and a `LinkedList`.

Lists

ListDemo.java

```
package examples.collections;
import java.util.*;
public class ListDemo {
    public static void main(String args[]) {
        List list = new ArrayList();
        for (int i = 0; i < args.length; i++)
            list.add(args[i]);
        System.out.println("ArrayList:");
        printList(list);
        list = new LinkedList();
        for (int i = 0; i < args.length; i++)
            list.add(args[i]);
        System.out.println("LinkedList:");
        printList(list);
    }
    public static void printList(Collection data){
        Iterator iter = data.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
        System.out.println();
    }
}
```

- Notice that elements are added to the `ArrayList` in the same way they are added to the `LinkedList`.
    - Notice also that the static `printList` method takes a `Collection` as its parameter.
    - Since an `ArrayList` and a `LinkedList` are a `Collection`, they can be passed as the argument.
        - The benefit of using a Collection as the parameter will be even clearer in the next example.
    - An `Iterator` acts similar to an `Enumeration`.
        - An `Iterator` has the added functionality of being able to remove an element.

Sets

- Set is a sub-interface of Collection in that it extends the Collection interface by stipulating that no duplicate elements are allowed.
  - The JDK provides two concrete implementations of Set.
    - HashSet - This class makes no guarantees as to the iteration order of its elements.
    - TreeSet - This class guarantees that the iteration order of its elements is in ascending order.
  - The example below demonstrates the behavior of both a HashSet and a TreeSet.

SetDemo.java

```
package examples.collections;
import java.util.*;
public class SetDemo {
    public static void main(String args[]) {
        Set uniqueWords = new HashSet();
        for (int i = 0; i < args.length; i++)
                uniqueWords.add(args[i]);
        System.out.println("HashSet:");
        ListDemo.printList(uniqueWords);
        uniqueWords = new TreeSet();
        for (int i = 0; i < args.length; i++)
                uniqueWords.add(args[i]);
        System.out.println("TreeSet:");
        ListDemo.printList(uniqueWords);
    }
}
```

- Note the use of the static printList(Collection data) method from the previous example to iterate through the elements in the each Set.
- Try running the above program passing several words on the command line, where several of the words are repeated.

Maps

- `Map` is another top-level interface of the Collections Framework.
  - As with the top-level `Collection` interface, the `Map` interface has several sub-interfaces and abstract classes below it in its hierarchy.
  - A `Map` is a collection of keys and values, where the keys are unique within the `Map`.

- Two of the concrete implementations of `Map` provided with the JDK are shown below.
  - `HashMap` - This class is the newer version of the `Hashtable` studied earlier in this section.
  - `TreeMap` - This class maintains the keys in sorted order.

- The example below demonstrates the behavior of a `TreeMap`.

MapDemo.java

```
package examples.collections;
import java.util.*;
public class MapDemo {
    public static void main(String argv[]) {
        Map cities = new TreeMap();
        cities.put("Richmond", "Virginia");
        cities.put("Boston", "Massachusetts");
        cities.put("Richmond", "Virginia");
        Set set = cities.keySet();
        Iterator iter = set.iterator();
        while (iter.hasNext())
            System.out.println(iter.next());
    }
}
```

The `Collections` Class

- The `Collections` class consists entirely of static methods that operate on or return collections.

- Here is an example demonstrating the use of `binarySearch` to search an ordered list.
    - This method returns the index where an element is found or a negative value if it does not exist in the list.
        - The actual negative value returned is:
          `(-(insertion point) - 1) = returnedValue`
          where `insertion point` indicates the point at which the element would be inserted into the list.

Search.java

```
package examples.collections;
import java.util.*;
public class Search  {
    public static void main(String args[]) {
        List list = new ArrayList();
        for (int i = 0; i <= 100; i += 2)  {
            Integer ival = new Integer(i);
            list.add(ival);
        }
        for (int i = 0; i < args.length; i++)  {
            Integer find = new Integer(args[i]);
            int pos =
                Collections.binarySearch(list, find);
            if (pos >= 0 )
                System.out.println(args[i] +
                    " found: at pos " + pos);
            else
                System.out.println(args[i] +
                    " not found");
        }
    }
}
```

Exercises

1. The lab files directory contains a file named `statecaps.txt` that lists all 50 states and their capitals.

   - Write a program that reads the file into a `HashMap` using the state as the key.
   - Obtain an `Iterator` from the `HashMap` to print out all 50 states and capitals.

2. Modify the previous exercise to enter a loop that randomly selects one of the 50 States from the `HashMap` and prompts the user to enter the capital.

   - The application should take the number of times to loop from the command line.
   - It should also keep track of how many times the user was correct out of the total number questions.

3. Create an application that stores all the numbers supplied as command line arguments and stores them in an `ArrayList`.

   - The program should print out the contents of the `ArrayList` two times.
     - The first time should use a `for` loop to loop by index.
     - The second time should rely on an `Iterator`.
   - Each loop should also print the total of all of the numbers.

This Page Intentionally Left Blank

Chapter 12: Networking

Networking Fundamentals

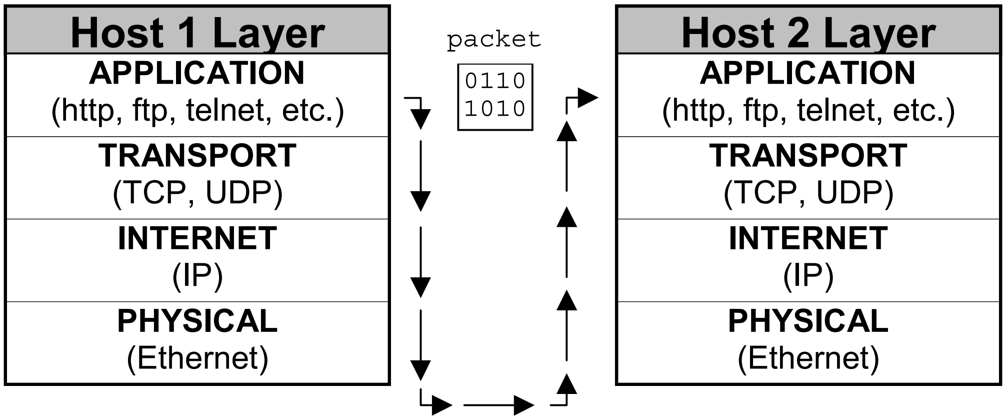
- Before looking at the examples of Java networking, we will give some basic details about networking in general.
  - Every machine on a network is called a node or a **host**. Each host has an **address,** which uniquely identifies it on the network.
    - The address is a four-byte number usually represented as four numbers separated by dots.

      ```
      127.0.0.1     192.168.0.100
      ```

    - Since humans are better at remembering names than numbers, a system has been developed to map names to network addresses. The **D**omain **N**ame **S**ystem (DNS) is incorporated into machines known as **Name Servers**.
  - Data is sent over a network in **packets**. Each packet contains the address of the sender and receiver.
  - Computers communicate through a set of rules called **protocols**.
    - Although there are many protocols in use, we will only be concerned with the **http** protocol and the **tcp** protocol.
  - When a process on one machine wishes to send a packet of information to a process on another machine, there are several layers through which the packet must travel.
    - These layers have been described by several models, most notably the seven layer model known as the **ISO/OSI** model.
    - Since we are interested in providing the networking details for Java programs, we will use a simplified version of the seven layer model.

Networking Fundamentals

- When a packet flows from **Host 1** to **Host 2**, the packet is sent through the layers as depicted in the diagram below.



- Each layer has distinct duties to perform.
  - The APPLICATION layer is usually the only one with which a Java programmer is concerned.
    - The application layer sends data from or delivers data to your program.
  - The TRANSPORT layer controls how the packets are delivered. The two most common protocols for this layer are described below.
    - **T**ransmission **C**ontrol **P**rotocol (TCP) - reliable, high overhead
    - **U**ser **D**atagram **P**rotocol (UDP) - unreliable, low overhead

- Each computer on a network can provide many networking services.
  - For example, your computer may provide a **F**ile**T**ransfer **P**rotocol (FTP) service and a **H**yper**T**ext **T**ransfer **P**rotocol (HTTP) service.

The Client/Server Model

- When an application on one machine needs a particular service on another machine, it is not enough for the application to reference the other machine's IP address alone.
  - It must reference the service on the machine it wishes to use.
  - Services are referenced by **port** numbers. Each service is mapped to a port. Standardized services use standard ports. For example:
    - `ftp` uses port 21;
    - `http` uses port 80; and
    - `telnet` uses port 23.
- Most networking applications today use the **Client/Server** model.
  - In this model, a process on one machine requests some service, such as a file service or time-of-day service from another process.
    - The process requesting the service is called the **client process**.
    - The process providing the service is called the **server process**.
  - The two processes could reside on the same machine, but more often, they reside on different machines.
    - The machine hosting the client process is called the **Client**.
    - The machine hosting the server process is called the **Server**.

The Client/Server Model

- One example of the Client/Server model follows.
  - Data is stored on a Server that services many requests from Client software hosted by PCs.
    - The Browser/Web Server fits the Client/Server model.
- Browsers and web servers communicate using the HTTP protocol.
  - The HTTP protocol defines how a client requests data from a server and how the data is transferred back to the client from the server.
    - A browser typically makes a request to the server for a file to retrieve. The name of the file and its location on the World Wide Web is specified as a **U**niform **R**esource **L**ocator (URL).
- A URL is a reference to a resource on the Internet. Most people are used to seeing URLs as Domain Names such as the following.

```
www.trainingetc.com     www.sun.com     www.apache.org
```

  - In reality, a domain name is just one part of a URL.
  - A URL can consist of the following parts, some of which are dependent upon the protocol.

• protocol://	`http://`
• hostname:port	`www.trainingetc.com:80`
• path	`courses/file#java`
• file	`courses/file`
• #section	`#java`

InetAddress

- The `java.net` package contains the majority of the classes that pertain to networking in Java.
  - The first class we will study in this package is the `InetAddress` class.
- `InetAddress` can be used to obtain information about a host name or an IP address.
  - The JVM will rely on the DNS server configured for the system to obtain the information.
  - The example on the next page calls various methods from the InetAddress class to display information about both the host name provided on the command line and the local host.

InetAddress

Address.java

```
package examples.networking;
import java.net.*;
public class Address {
    public static void main(String args[])  {
        try {
            print("Remote Host:");
            InetAddress remote =
                InetAddress.getByName(args[0]);
            getInfo(remote);
            print("Local Host:");
            InetAddress local =
                InetAddress.getLocalHost();
            getInfo(local);
        } catch(UnknownHostException e) {
            print("???: " + e.getMessage());
        }
    }
    private static void getInfo(InetAddress ia){
        print("  HostName: " +
              ia.getHostName());
        print("  HostAddress: "
              + ia.getHostAddress());
        print("  CanonicalHostname: " +
              ia.getCanonicalHostName());
        getRawIP(ia);
        print("");
    }
    private static void getRawIP(InetAddress ia){
        byte [] b = ia.getAddress();
        System.out.print("  ");
        for (int i = 0; i < b.length; i++) {
            int each = b[i] < 0 ? b[i] + 256 : b[i];
            System.out.print(each + " ");
        }
        System.out.println();
    }
    private static void print(String s){
        System.out.println(s);
    }
}
```

URLs

- `URL` is another class in the `java.net` package.
  - A URL can be constructed in various ways, several of which are shown below.

```
public URL(String spec) throws MalformedURLException

public URL(String protocol, String host,
           String file) throws MalformedURLException

public URL(String protocol, String host, int port,
           String file) throws MalformedURLException
```

  - Once a `URL` is constructed, several methods can be used to retrieve a specific field as shown in the example below.
    - The application below requires that a host name be supplied on the command line.

URLS.java

```
package examples.networking;
import java.net.*;
public class URLS {
    public static void main(String args[]) {
        try {
            URL u = new URL("http", args[0], 80,
                            "/index.html");
            System.out.println(u);
            print("Prot: " + u.getProtocol());
            print("Host: " + u.getHost());
            print("Port: " + u.getPort());
            print("Ref:  " + u.getRef());
            print("File: " + u.getFile());
        } catch(MalformedURLException e) {
            System.out.println(e.getMessage());
        }
    }
    private static void print(String s){
        System.out.println(s);
    }
}
```

URLs

- The `openStream` method in the `URL` class returns an `InputStream` that can be used to read the data at the given `URL,` as shown in the example below.

ReadWebPage.java

```
package examples.networking;
import java.net.*;
import java.io.*;
public class ReadWebPage {
    public static void main(String args[]) {
        try {
            URL web = new URL("http://" + args[0]);
            InputStream  is = web.openStream();
            int aByte = 0;
            while((aByte = is.read()) != -1)
                System.out.print((char) aByte);
            is.close();
        } catch(MalformedURLException e)  {
            System.out.println("Malformed");
        } catch(IOException e) {
            System.out.println("IOException");
        }
    }
}
```

- Below is some sample output from the above program.

```
java examples.networking.ReadWebPage
www.trainingetc.com
<html>
<head>
<title>/training/etc Technical Training</title>
<meta http-equiv="Content-Type" content="text/html;">
<!-- Fireworks MX Dreamweaver MX target.  Created Thu
Mar 31 13:56:00 GMT-0500 (Eastern Standard Time) 2005-
->
```

URLs

- A `URLConnection` can be obtained from the `openConnection` method of the `URL` class.
  - The `URLConnection` class has various methods to obtain the header information sent by the server instead of the body of the response from the server as in the previous example.
    - Certain header fields that tend to be accessed frequently have special methods to obtain them.
  - The example below demonstrates the use of the `URLConnection` to obtain information about the headers sent by the server.

ReadHeaders.java

```
package examples.networking;
import java.net.*;
import java.io.*;
import java.util.*;
public class ReadHeaders {
    public static void main(String argv[]) {
        try {
            URL u = new URL("http://" + argv[0]);
            URLConnection uc = u.openConnection();
            print("Content Type:" +
                uc.getContentType());
            print("Content Length:" +
                uc.getContentLength());
            print("Date:" + new Date(uc.getDate()));
            print("Last Modified:" +
                new Date(uc.getLastModified()));
        } catch(MalformedURLException e) {
            System.out.println("Malformed");
        } catch(IOException e) {
            System.out.println("IOException");
        }
    }
    private static void print(String s){
        System.out.println(s);
    }
}
```

Sockets

- Now, we will show a few examples of Clients and Servers that communicate.
  - This typically involves the use of the `Socket` and `ServerSocket` classes.
- A `Socket` is an endpoint of a link between two programs running on a network.
  - A `Socket` uses a port number to identify the application to which the TRANSPORT layer should send the data as it arrives (or is delivered) over the internet.
    - The `Socket` class is used by Clients and Servers to communicate over the network.
  - A typical Client application needs to:
    - connect to a remote machine;
    - send and receive data over the connection; and
    - close the connection.
  - A typical Server application needs to:
    - bind to a port;
    - listen for a connection;
    - accept connections on the port;
    - send and receive data over the connection; and
    - close the connection.
- The `Socket` class has several constructors, most of which take an address and a port for their first two parameters.

Sockets

- The example below is a simple Client that attempts to connect to a Server listening for connections on port 80.
  - The constructor for the `Socket` will need to obtain the host name to connect to from the command line.
  - The application simply demonstrates several of the methods available in the `Socket` class, which allow a developer to obtain information about the `Socket`.

SocketInfo.java

```java
package examples.networking;
import java.net.*;
import java.io.*;
public class SocketInfo {
    public static void main(String args[]) {
        try {
            Socket s = new Socket(args[0], 80);
            print("Connected:")
            print("To: " + s.getInetAddress());
            print("  on port  " + s.getPort());
            print("From " + s.getLocalAddress());
            print("  on port " + s.getLocalPort());
            s.close();
        } catch(UnknownHostException e) {
            print("Unknown Host: " + args[0]);
            e.printStackTrace();
        } catch(SocketException e) {
            print("SocketException: " + args[0]);
            e.printStackTrace();
        } catch(IOException e) {
            print("IOException:");
            System.out.println(e);
            e.printStackTrace();
        }
    }
    private static void print(String s){
        System.out.println(s);
    }
}
```

A Time-of-Day Client

- We will now show an example of a client that connects to a Daytime service, which is a service supplied by many Servers on port 13.
  - In order to do this, we need to know how to read and write with sockets.
  - The `Socket` class has a pair of methods that will allow communication.

    ```
    public InputStream getInputStream() throws IOException
    public OutputStream getOutputStream() throws IOException
    ```

  - In the example below, data will be flowing in only one direction (from the Server to the Client). Therefore, we only need to rely on the `getInputStream` method of the `Socket`.

DayTimeClient.java

```
package examples.networking;
import java.net.*;
import java.io.*;
public class DayTimeClient {
    public static void main(String args[]) {
        String host = "time.nist.gov";
        try {
            if(args.length > 0)
                host = args[0];
            Socket s = new Socket(host, 13);
            InputStream is = s.getInputStream();
            System.out.println("Time at " +
                                    host + " is");
            int data;
            while((data = is.read()) != -1)
                System.out.print((char) data);
            is.close();
            s.close();
        }catch(IOException e) {
            System.out.println(e);
        }
    }
}
```

Writing Servers

- Before we look at a Java Client application communicating with a Java Server application, we need to know more about how Servers are written in Java.

- The `ServerSocket` class is similar to the `Socket` class. However, it provides additional methods in support of the extra tasks for which a server is typically responsible.

- A server typically has the following life cycle.
  - Bind to a particular port during its construction.
  - Listen for a Client on that port by calling `accept()` from the `ServerSocket` class.
  - Use the `Socket` returned from the `accept` method to then call the `getInputStream` and/or `getOutputStream` method(s) on the `Socket`.
  - The business of the Client and the Server is then transacted.
  - The connection is closed by either the Client or the Server.
  - Typically, the Server then waits for another connection.

- Several forms of the `ServerSocket` constructor are shown below.
  - `public ServerSocket(int port) throws IOException` binds the Server to the specified port.
  - `public ServerSocket(int port, int backlog) throws IOException` binds the Server to the specified port and sets the maximum queue length for incoming, pending connections.

Client/Server Example

- The following example demonstrates an "Echo Server," which echoes, in upper case, whatever data it receives back to the client.
  - Comments have been placed in the code to indicate the various aspects of the life-cycle of the Server.

EchoServer.java

```
package examples.networking;
import java.net.*;
import java.io.*;
public class EchoServer {
    public static void main(String args[]) {
        ServerSocket theServer = null;
        Socket clientSocket;
        int port = 2345;
        InetAddress ia = null;
        // Attempt to start the server
        // bound to the given port
        try{
            theServer = new ServerSocket(port);
            // Print info about the server
            ia = InetAddress.getLocalHost();
            String host = ia.getHostAddress();
            System.out.println("Server started on " +
                host+ "  Listening on port "+ port);
            // loop for each client
            while(true){
                // wait for a client to connect
                clientSocket = theServer.accept();
                // handle client in a helper method
                handleClient(clientSocket);
            } // proceed to next Client
        } catch(IOException ioe){
            ioe.printStackTrace();
            System.exit(1);
        }
    }
```

- Code continued on following page

Client/Server Example

EchoServer.java - continued

```
    // Helper method to handle client communications
    private static void handleClient(Socket cSocket){
         System.out.println(cSocket.getInetAddress()
            + ":Connected");
        PrintStream toClient;
        BufferedReader fromClient;
        String data;
        try{
            // Get Input and Output
            fromClient = new BufferedReader(
                new InputStreamReader(
                    cSocket.getInputStream()));
            toClient = new PrintStream(
                cSocket.getOutputStream());
            while(true){
                // read from Client
                data = fromClient.readLine();
                if(data == null) break;
                data = data.toUpperCase();
                // write to Client
                toClient.println(data);
            }
            fromClient.close();
            toClient.close();
            cSocket.close();
        }catch(IOException ioe){
            String msg = "Connection lost";
            System.out.println(msg);
        }finally{
            System.out.println(
                cSocket.getInetAddress() +
                ":DisConnected");
        }
    }
}
```

- A Client application that is capable of communicating with the above Server is shown on the next page.

Client/Server Example

EchoClient.java

```
package examples.networking;
import java.net.*;
import java.io.*;
public class EchoClient {
    public static void main(String args[]) {
        Socket con = null;
        PrintStream toServer;
        BufferedReader fromServer, fromKB;
        String data;
        int port = 2345;
        String host = "localhost";
        if(args.length > 0)
            host = args[0];
        try{
            // Attempt to connect to server
            con = new Socket(host, port);
        } catch(IOException ioe){
            // No use in continuing
            String msg = "Unable to connect";
            System.out.println(msg);
            ioe.printStackTrace();
            System.exit(1);
        }
        try{
            // get Input(s) and Output
            fromKB = new BufferedReader(
                new InputStreamReader(
                    System.in));
            fromServer = new BufferedReader(
                new InputStreamReader(
                    con.getInputStream()));
            toServer = new PrintStream(
                con.getOutputStream());
            // communicate with the server
            String prompt = "Enter Data:\n" +
                "Entering just the word QUIT will " +
                "Close the connection.";
            System.out.println(prompt);
```

- Code continued on following page

Client/Server Example

EchoClient.java - continued

```
            while(true){
                // read from keyboard
                data = fromKB.readLine();
                if(data.equals("QUIT")) break;
                // write data to server
                toServer.println(data);
                // read response from server
                System.out.println(
                    fromServer.readLine());
            }
            // close resources
            fromServer.close();
            toServer.close();
            con.close();
        }catch(IOException ioe){
            String msg = "Connection lost";
            System.out.println(msg);
        }
    }
}
```

- To test the above application, open a separate DOS window for the `EchoServer` and one DOS window for each `EchoClient`.
    - The server, as written, is only able to handle one client at a time.
        - This is because the while loop that the server uses does not advance to the next iteration until the `handleClient` method returns, which enables the server to wait for another client by calling `accept` again.
        - The work currently performed by the handleClient method is a perfect candidate for a thread.

Exercises

1. The `DayTimeClient` application from this chapter relied on a pre-existing Daytime service to be available.

   - Write your own version of a `DayTimeServer` that is capable of handling the current `DayTimeClient`.

2. Working with copies of both the `DayTimeServer` and `DayTimeClient`, complete the following exercise.

   - Modify the `DayTimeServer` so that it writes a `Date` object to the client (using an `ObjectOutputStream`) rather than the date as a `String`.

   - Modify the `DayTimeClient` to read a `Date` object from the server (using an `ObjectInputStream`) rather than the date as a `String`.

This Page Intentionally Left Blank

Chapter 13: Threads

Threads vs. Processes

- To learn about threads, one has to retreat to some fundamental computer terminology.
  - A computer is composed of several distinct parts, one of which is the **C**entral **P**rocessing **U**nit (**CPU**).
  - When a program is executing, instructions from the program are fetched in sequence from memory into the CPU for processing. The program in execution is called a **process**.
  - It is usually said that a modern computer can execute many processes concurrently. This is sometimes erroneously called **multi-processing**.
    - In reality, many processes are in various states of execution in any single instant.
    - However, when a computer has a single CPU, then in any given instant, only a single instruction may be executed. Therefore, true multi-processing can only be achieved when a computer has more than one processor.
    - On a single CPU computer, the fact that many processes are in various states of execution at a single instant is known as **multi-programming**. Each process has its own variables.
  - The ability of a single process to spawn multiple execution paths is called **multi-threading**. Each path is called a **thread**. A thread is also referred to as a lightweight process.
    - Unlike a process, each thread shares the same set of data (variables). If two threads access this data at the same time, there can be synchronization problems.
    - Generally, multi-threading is a blessing since it allows the same program to handle multiple events "concurrently."
  - There are two ways to create a new thread of execution.
    - Declare a class that extends `Thread`.
    - Declare a class that implements the `Runnable` interface.

Creating Threads by Extending `Thread`

- The `java.lang.Thread` class implements the `Runnable` interface.
  - Therefore, one way of creating a thread is to create a class that extends `Thread` and overrides the `run` method.

- When a `Thread` is created it is in the new state.
  - It does not enter the runnable state until it is started.
  - A `Thread` is started by calling its `start` method.
    - This notifies the thread scheduler that a new thread can now be started at some time in the near future.

MyThread.java

```
package examples.threads;
public class MyThread extends Thread {
    public MyThread(String s) {
        super(s);
    }
    public void run() {
        for(int i = 0; i < 5; i++){
            System.out.println(getName() + " " + i);
        }
    }
    public static void main(String a[]) {
        MyThread t;
        t = new MyThread("Thread A");
        t.start();
        t = new MyThread("Thread B");
        t.start();
        for (int i = 0; i < 5; i++)
          System.out.println("MainThread " + i);
    }
}
```

- There are three threads in the code above, the main thread and the two `MyThread` threads.

Creating Threads by Extending `Thread`

- Each thread is started with the following two lines of code.

  ```
  t = new MyThread("Thread A");
  t.start();
  ```

  - The `start` method does not operate as a "normal" method.
    - The `start` method makes the thread runnable and also returns immediately.
  - When the thread scheduler decides to run each `MyThread` object, it does so by calling its `run` method.
    - Therefore, the `run` method of a `Thread` can be viewed as being similar to the `main` method of an application.
  - When the `start` method returns for each `MyThread,` each thread competes for CPU time with the `main` method.

- The static `Thread.currentThread` method returns a reference to the currently executing thread object.

- It is not always possible to use the approach of extending the `Thread` class, because the class may already extend another class.
  - Java does not permit multiple inheritance of implementation.
  - For example, all applets must extend the `Applet` class.
    - Therefore, having your applet extend both `Applet` and `Thread` is not allowed.
    - There is a second technique used to create a thread that consists of implementing the `Runnable` interface.

Creating Threads by Implementing `Runnable`

- The `Runnable` interface defines a single method named `run` as shown below.

```
public interface Runnable {
    public void run();
}
```

  - When you implement the `Runnable` interface, you still must create a `Thread` object by passing a reference to your `Runnable` object to the `Thread` objects constructor.
  - The process is demonstrated in the code below.

MyRunnable.java

```
package examples.threads;
public class MyRunnable implements Runnable {
    public MyRunnable() { }
    public void run() {
        String name =
            Thread.currentThread().getName();
        for (int i = 0; i < 5; i++)
            System.out.println(name + " " + i);
    }
    public static void main(String args[]) {
        Thread t;
        t = new Thread(new MyRunnable());
        t.start();
        t = new Thread(new MyRunnable());
        t.start();
        String name =
            Thread.currentThread().getName();
        for (int i = 0; i < 5; i++)
            System.out.println(name + " " + i);
    }
}
```

  - When the thread scheduler decides to run each `Thread` object, it does so by calling the `run` method from the `Runnable` object passed to the `Thread` constructor.

Advantages of Using Threads

- There are several reasons to use threads. Threads can:
    - isolate tasks and make programs easier to follow;
    - make your program run faster; and
    - be used as progress indicators of another thread running in the background.

- In order to understand the uses of threads and the advantages they offer, we will demonstrate several versions of an application that rely upon the `copy` method in the class shown below.

`FileCopyUtility.java`

```
package examples.threads;
import java.io.*;
public class FileCopyUtility{
    public static void copy(File src, File dest){
        if (!src.isDirectory()){
            FileInputStream fis = null;
            FileOutputStream fos = null;
            try{
                fis = new FileInputStream(src);
                fos = new FileOutputStream(dest);
                int theByte;
                while( (theByte = fis.read()) != -1){
                    fos.write(theByte);
                    // Simulate large file being read
                    try{Thread.sleep(10);}
                    catch(InterruptedException ie){}
                }
                fis.close();
                fos.close();
            } catch(IOException ioe){
                ioe.printStackTrace();
            }
        }
    }
}
```

Advantages of Using Threads

- The application below copies a list of files whose names are supplied on the command line.
  - This version of the application does not use threads.
    - During each copy process, there is no onscreen indication of the progress, which may result in a user wondering if anything is actually happening.

FileCopier1.java

```
package examples.threads;
import java.io.*;
public class FileCopier1 {
    public static void main(String args[]){
        for(int i = 0; i < args.length; i++){
            File source = new File(args[i]);
            File dest =
                new File ("C:/javalabs/" + args[i]);
            System.out.println();
            System.out.println("Copying " + args[i]);
            FileCopyUtility.copy(source, dest);
        }
    }
}
```

- The example on the next page uses a thread as a progress indicator to provide more feedback to the user during the copying process.

Advantages of Using Threads

FileCopier2.java

```java
package examples.threads;
import java.io.*;
public class FileCopier2 {
    public static void main(String args[]) {
        Thread t = new ProgressIndicator();
        t.start();
        for(int i = 0; i < args.length; i++){
            File source = new File(args[i]);
            File dest =
                new File ("C:/javalabs/" + args[i]);
            System.out.println();
            System.out.println("Copying " + args[i]);
            FileCopyUtility.copy(source, dest);
        }
        t.interrupt();
    }
}
```

ProgressIndicator.java

```java
package examples.threads;
public class ProgressIndicator extends Thread{
    public void run() {
        while(true){
            System.out.print('.');
            try{
                Thread.sleep(1000);
            }catch(InterruptedException ie){
                break;
            }
        }
    }
}
```

- Calling the `interrupt` method of the thread results in the thread breaking out of the loop, terminating the application.

Daemon Threads

- Threads can be categorized as either "**user**" or "**daemon**" threads.

- The JVM will continue to run as long as the thread scheduler has at least one "user" thread that is running.
  - The `main` method of an application runs in a user thread, and if no additional threads are created by the program, the JVM terminates when the end of the `main` method is reached.
    - The application on the previous page created an additional user thread that looped forever, unless the thread was interrupted.

- A daemon thread is normally a low priority thread that runs in the background.
  - The JVM will terminate if the only running threads are daemon threads.
    - The garbage collector is an example of a daemon thread.
  - The example on the following page calls the `setDaemon` method on the thread prior to calling its `start` method.
    - This means that once the `main` method has completed, the only remaining thread will be a daemon thread.
    - Therefore, the JVM will terminate without our code needing to interrupt the thread, as in the previous example.

Daemon Threads

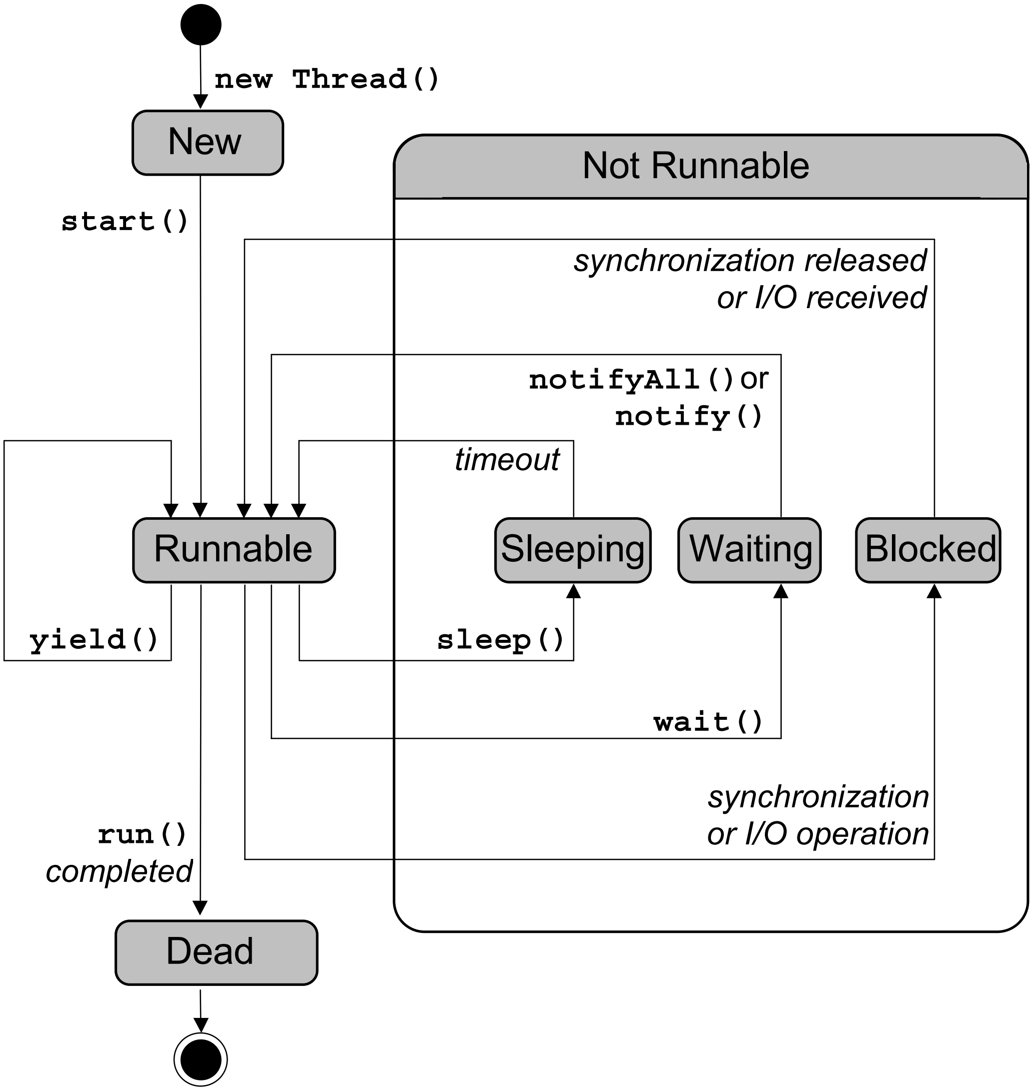
FileCopier3.java

```
package examples.threads;
import java.io.*;
public class FileCopier3 {
    public static void main(String args[]) {
        Thread t = new ProgressIndicator();
        t.setDaemon(true);
        t.start();
        for(int i = 0; i < args.length; i++){
            File source = new File(args[i]);
            File dest =
                new File ("C:/javalabs/" + args[i]);
            System.out.println();
            System.out.println("Copying " + args[i]);
            FileCopyUtility.copy(source, dest);
        }
    }
}
```

- When the `for` loop above is completed, the `main` method (which runs in a user thread) will have completed.
    - Since the only remaining thread (`t`) was started up as a daemon thread by calling the `setDaemon` method, the JVM will terminate even though thread (`t`) has not finished.

Thread States

- In any instant, a thread can be in one of various states during its lifetime.
  - The possible thread states are shown in the diagram below.
    - **new** - if the constructor has been called
    - **runnable** - if its `start` method has been called
    - **not runnable** - (blocked) if its `sleep` or `wait` method has been called, or it is blocking on I/O or `synchronized` code
    - **dead** - if its `run` method has completed



Thread States

- Some `Thread` methods can be executed only in certain states. An `IllegalThreadStateException` will be thrown otherwise.
  - An example of this would be trying to call that start method on a thread that is in the dead state.
    - This implies that a thread can only be run once.
- The `join` method will cause a thread to wait for the thread the method is called on to die.
- The `yield` method will cause a thread to yield its time to another thread of the same priority.
- The example on the next page will demonstrate some of the methods that control the state of a `Thread`.
  - The application asks the user for keyboard input.
  - If the user does not respond within a certain amount of time, the program will terminate.
    - The timing will be handled by a thread.

Thread States

TimerThread.java

```
package examples.threads;
public class TimerThread implements Runnable {
    int secs;
    public TimerThread(int s) {
            secs = s;
    }
    public void run() {
        System.out.print("Timer set for ");
        System.out.println(secs + " seconds.");
        try {
            Thread.sleep(secs * 1000);
        }
        catch(InterruptedException e) {
            System.out.print("Timer Thread ");
            System.out.println("Interrupted");
            return;
        }
        System.out.println("You are too slow ...");
        System.exit(0);
    }
}
```

- The `TimerThread` above is designed to sleep for a certain number of seconds and terminate the program when it is done sleeping.
    - The only way this thread will not terminate the program that it is running within is if the thread is interrupted.

- The application on the next page contains a loop that:
    - creates an instance of the `Runnable` class defined above;
    - passes the reference to a `Thread` constructor and starts the `Thread`; and
    - requests user input from the keyboard.

Thread States

TimedDataEntry.java

```
package examples.threads;
import java.io.*;
public class TimedDataEntry {
    public static void main(String args[])
      throws IOException{
        String str;
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        TimerThread timer;
        Thread t;
        while(true) {
            timer = new TimerThread(10);
            t = new Thread(timer);
            t.start();
            System.out.println("Enter a string: ");
            str = br.readLine();
            t.interrupt();
            System.out.println("You entered " + str);
        }
    }
}
```

- Each time a line of text is successfully read from the keyboard, the `interrupt` method is called to prevent the timer thread from terminating the application.

Thread Problems

- The example on the next page demonstrates some of the problems that can occur when using threads in an application.
  - The application loops through several `int` arrays that are stored in a two dimensional array.
  - Each loop creates a thread to handle the sorting and printing of the contents of one of the `int` arrays.
    - The intended output is to see each array output in sorted order, although which order each array appears in the output does not matter.
  - The static `print` method in the class below will be used to do the actual sorting and printing.

PrintingUtils.java

```java
package examples.threads;
import java.util.*;
public class PrintingUtils{
    public static void print(int x[]){
        Arrays.sort(x);
        for(int i = 0; i < x.length; i++){
            System.out.print(x[i]);
            if(i < x.length - 1)
                System.out.print(",");
        }
        System.out.println();
    }
}
```

Thread Problems

- In the example below, each `Thread` (`SyncProblems`) is composed of an `int` array that will be passed to the `PrintingUtils.print` method.

SyncProblems.java

```
package examples.threads;
public class SyncProblems extends Thread{
    public static void main(String args[]){
        int odds [][] = {{9, 8, 7, 6, 5, 4, 3, 2, 1},
                  {52, 22, 32, 72}, {43, 83, 63, 3},
                  {24, 94, 54, 84}, {15, 65, 85, 5},
                  {36, 26, 66, 56}, {97, 17, 37, 7}};
        Thread t1;
        for(int i = 0; i < odds.length; i++){
            t1 = new SyncProblems(odds[i]);
            t1.start();
        }
    }
    int a [];
    public SyncProblems(int a []){
        this.a = a;
    }
    public void run(){
        PrintingUtils.print(a);
    }
}
```

- Although the original intent was to display each array sorted, the output of the above application is shown below.

```
1,2,3,4,223245267,,,,,,324354153617,,,,,,526384655637,
,,5,,,7283,946685

6

,7,8,9
97
```

- The next page introduces the `synchronized` keyword as a means of correcting the problem.

Synchronization

- The problem in the previous code was that multiple threads were inside of the `PrintingUtils.print` method at the same time, each competing for access to the standard output `System.out` to print the array.

- To guard against this, every object in Java has a lock with which it is associated.
  - When an object is locked by one thread, and another thread tries to call a `synchronized` method or `synchronized` block on the same object, the second thread will block until the object is unlocked.
  - The portion of the code that is `synchronized` is often referred to as a **critical section**.
  - The example below is a revised version of the previous `PrintingUtils.java,` where the `print` method has been `synchronized`.

`PrintingUtils2.java`

```java
package examples.threads;
import java.util.*;
public class PrintingUtils2 {
    public static synchronized void print(int x[]){
        Arrays.sort(x);
        for(int i = 0; i < x.length; i++){
            System.out.print(x[i]);
            if(i < x.length - 1)
                System.out.print(",");
        }
        System.out.println();
    }
}
```

  - The application to test the new version of the `print` method is shown on the next page.

Synchronization

NoSyncProblems.java

```
package examples.threads;
public class NoSyncProblems extends Thread{
    public static void main(String args[]){
        int odds [][] = {{9, 8, 7, 6, 5, 4, 3, 2, 1},
                {52, 22, 32, 72}, {43, 83, 63, 3},
                {24, 94, 54, 84}, {15, 65, 85, 5},
                {36, 26, 66, 56}, {97, 17, 37, 7}};
        Thread t1;
        for(int i = 0; i < odds.length; i++){
            t1 = new NoSyncProblems(odds[i]);
            t1.start();
        }
    }
    int a [];
    public NoSyncProblems(int a []){
        this.a = a;
    }
    public void run(){
        PrintingUtils2.print(a);
    }
}
```

- The output generated by the above application is shown below.

```
1,2,3,4,5,6,7,8,9
22,32,52,72
3,43,63,83
24,54,84,94
5,15,65,85
26,36,56,66
7,17,37,97
```

- Although synchronizing the method resulted in the desired output, the sorting process inside of the method was never really a problem (the inconsistent output came from the printed results).
    - For this reason, it would have been more efficient to make each thread block only for the printing process - not the sorting.

Synchronization

- A synchronized block of code can be used to surround a critical region of code, rather than the entire method.

- The example below shows a new version of the `PrintingUtils` that only synchronizes the printing, allowing each thread to be sorted prior to being blocked.

PrintingUtils3.java

```
package examples.threads;
import java.util.*;
public class PrintingUtils3 {
    static Object o = new Object();
    public static void print(int x[]){
        Arrays.sort(x);
        synchronized (o){
            for(int i = 0; i < x.length; i++){
                System.out.print(x[i]);
                if(i < x.length - 1)
                    System.out.print(",");
            }
            System.out.println();
        }
    }
}
```

  - The code above obtains the lock associated with the `Object o` to synchronize the critical region.
  - Since the `print` method itself is no longer `synchronized`, each thread is able to enter the method and execute the `sort` method prior to being blocked by another thread that may have already entered the `synchronized` block of code.
  - When a thread leaves the `synchronized` block of code, the lock it holds is relinquished, and another thread is able to enter the region and obtain the lock from `Object o`.

Exercises

1. Write a class `LetterThread`, a subclass of `Thread`, and create a few instances of it.

    - Each thread will print a `letter` of the alphabet `x` amount of times.

        - Both `letter` and `x` are given as arguments to the `LetterThread` constructor.

    - Inside the `run` method of your `LetterThread` class, compute a random number between 250 and 750 and use this as the number of ms to sleep inside your loop.

    - Run your program several times and notice the variety of outputs.

2. Starting with a copy of `LetterThread`, modify the new version so that instead of subclassing the `Thread` class, this rewrite should be a class which implements the `Runnable` interface.

3. Modify a copy of the `EchoServer` to be a concurrent server using threads to handle each client.

    - Also, include in the server output the address of each client connecting and an indication of when the client disconnects.