Introduction to Computers and Programming Languages

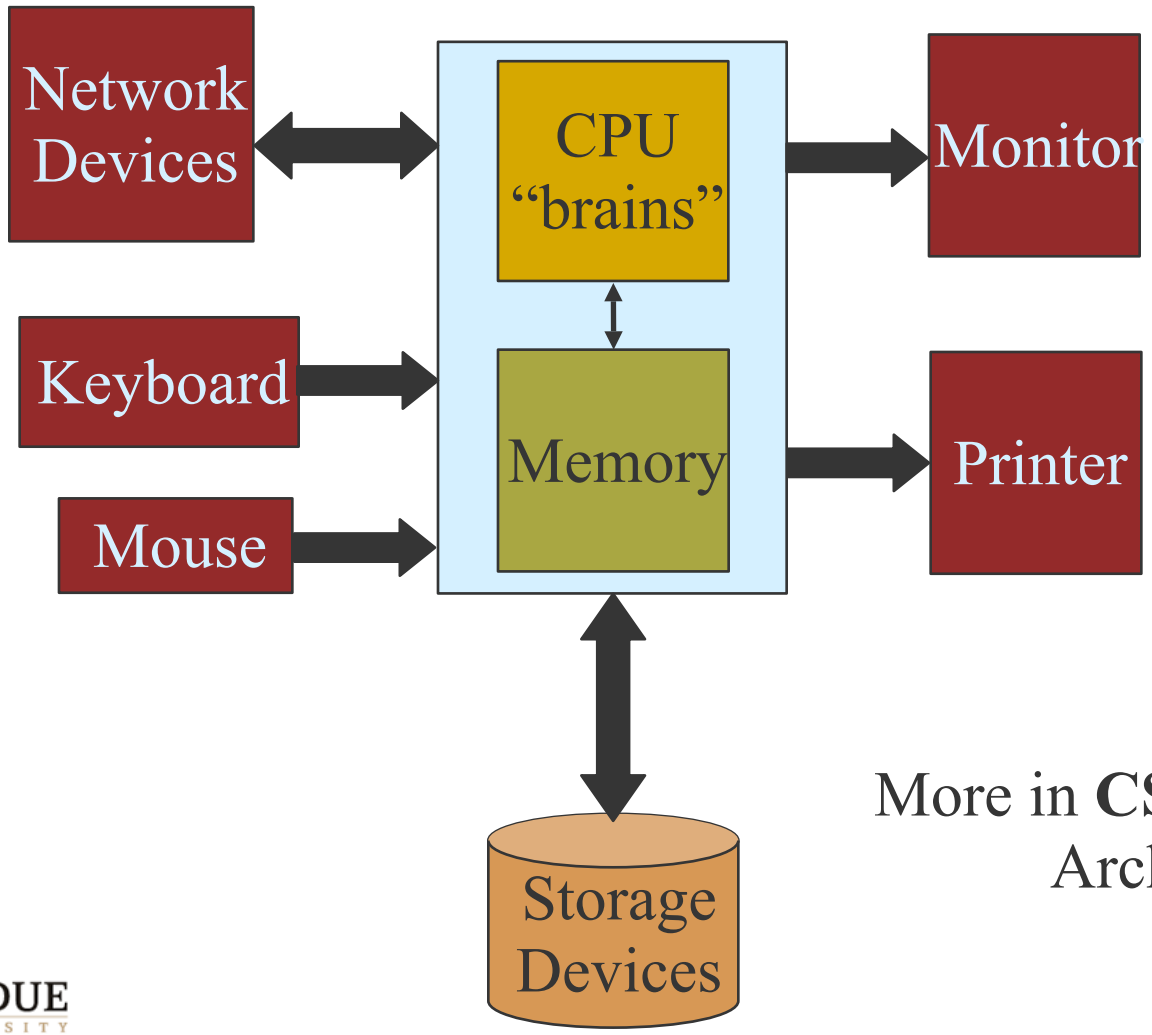
CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University

Week 1 Objectives

This week we will REVIEW:

- Computer systems and Java
- Simple Java programs
- Java data types
- Conditional statements: if and switch

Computer Architecture (simplified)



More in **CS250**: Computer Architecture.

Software

- Everything is in binary -- 0s and 1s
- Two types of information
  - Instructions(programs) -- executed by the CPU
  - Data -- manipulated by CPU
- These are stored in memory
- The software provides a means to access and control the hardware
- This is done through a very important piece of software called the Operating System
- The OS is always running. More in **CS354**

Machine Language

- A computer only runs programs that are specified in its own machine language (ML)
- Also called binary or executable code.
- The ML is specific to the CPU, e.g. Pentium, 386, PowerPC G3, G4, …
- A program written for one CPU will not run on another CPU -- i.e. it is not portable.

Assembly language

- Machine language codes are not easy to remember
- Assembly language uses mnemonics and symbols to ease programming, e.g.

  JMP L2
- A special program called an assembler must be used to convert the assembly code to machine code
- The assembly code is also hardware-specific.
- Eases programming but still requires one to think in terms of low-level steps taken by the CPU.
- Humans think at a higher level.

High-Level Languages

- Allow programmers to work with constructs that are closer to human language.
  - E.g. Java, C, C++, Basic, Fortran, COBOL, Lisp, …
- Need a special purpose program to convert the high-level program to machine language.
- This program is called a compiler.
- Can write programs in many different HLLs for the same CPU.
- Need a compiler for each language and CPU (OS).
- Efficient conversion is still an issue. More in **CS352** Compilers

High-Level Languages (cont.)

- Since the language is not specific to the hardware, HLL programs are more portable
  - Some hardware, OS issues limit portability
- All we need is the program and a compiler for that language on the given hardware platform
  - E.g. a C compiler for Mac OSX
- Thus we can write a program once in a HLL and compile it to run on various platforms, e.g. Netscape

Algorithms

- Humans tend to think of programs at a higher level than HLL -- more in terms of algorithms.
- An algorithm is a well-defined, finite set of steps that solves a given problem
  - E.g. the rules for multiplying two numbers

HLL Paradigms

- Procedural
  - A program is composed of packets of code called procedures, and variables. A procedure is at full liberty to operate on data that it can see. E.g. C, Pascal, COBOL, Fortran
- Object-Oriented
  - Programs are composed of Objects, each of a specific class with well defined methods. Data and programs are tightly coupled -- better design. E.g. Java, C++, Objective-C, C#
- Functional
  - Programs are composed of functions. E.g. Lisp
- More in **CS456** Programming Languages.

Object-Oriented Programming

- The OOP paradigm uses the notion of objects and classes as basic building blocks
- Other important components of OOP are
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Dynamic binding

Java

- Java is based upon C++ (which in turn is based on C).
- Unlike C++ which is really a hybrid language, Java is purely Object-Oriented.
- This results in significant advantages.
- Most HLL programs are compiled to run on a single platform.
- Java programs can run on multiple platforms after compilation -- i.e. its compiled format is platform-independent.
- This design choice comes from its history.

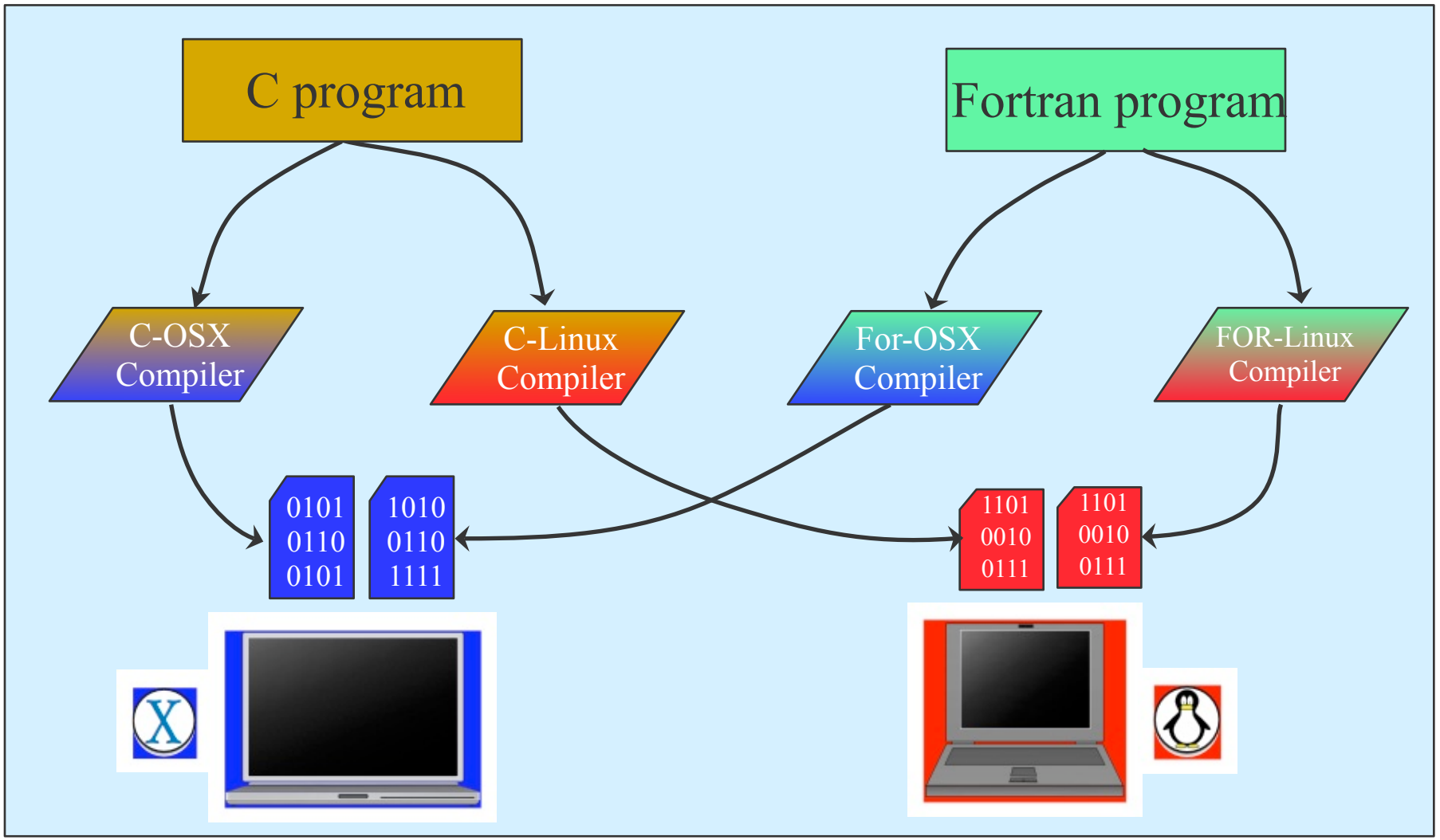
History of Java

- Java was developed by J. Gosling at Sun Microsystems in 1991 for programming home appliances (variety of hardware platforms).
- With the advent of the WWW (1994), Java's potential for making web pages more interesting and useful was recognized. Java began to be added to web pages (as applets) that could run on any computer (where the browser was running).
- Since then it has been more widely accepted and used as a general-purpose programming language, partly due to
  - its platform-independence, and
  - it is a truly OO language (unlike C++)

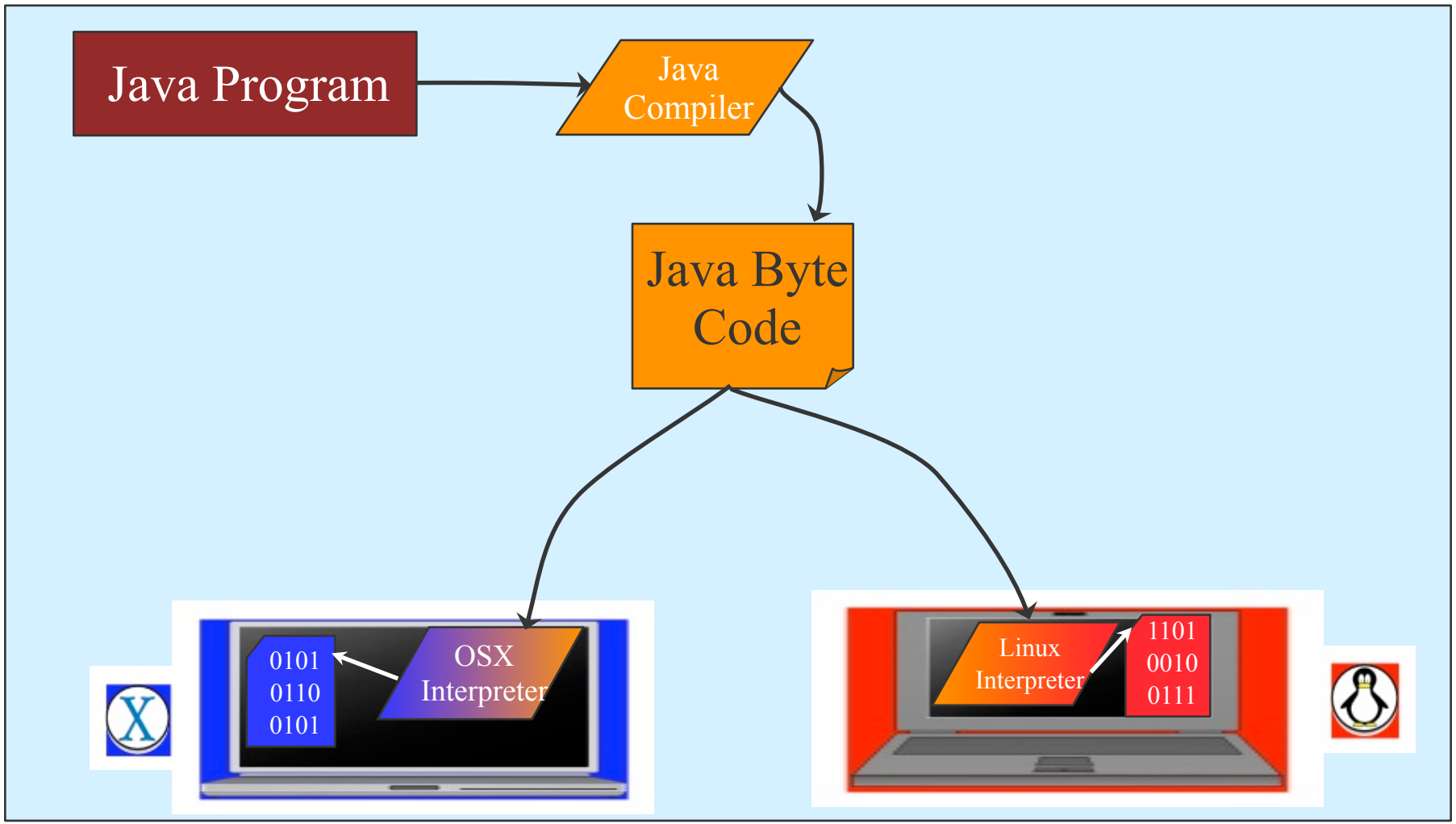
Platform-Independence

- Notion of a “Java Virtual Machine” (JVM)
- Java programs are compiled to run on a virtual machine (just a specification of a machine). This code is called Byte Code
- Each physical machine that runs a Java program (byte code) must “pretend” to be a JVM.
- This is achieved by running a program on the machine that implements the JVM and interprets the byte code to the appropriate machine code.
- This interpreting is done at run-time which can cause a slow down!

Regular Programming Languages



Java

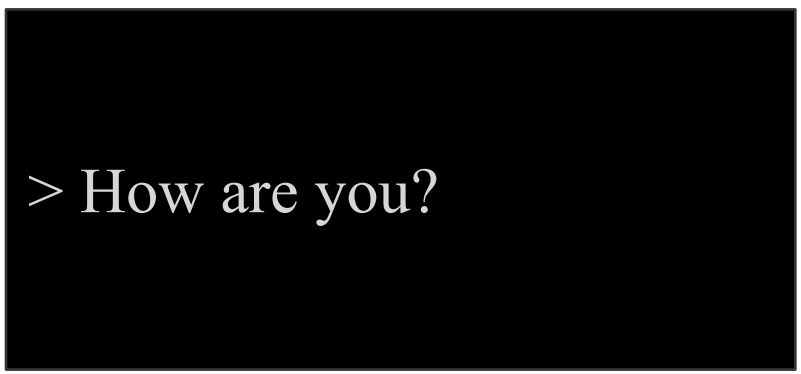

Java Program
Java Compiler
Java Byte Code
0101
0110
0101
OSX Interpreter
Linux Interpreter
1101
0010
0111

Simple Input & Output

Standard Output

- Using the **print** method of the **System.out** class is a simple way to write to the console window from which the program was run.

```
System.out.print("How are you?");
```

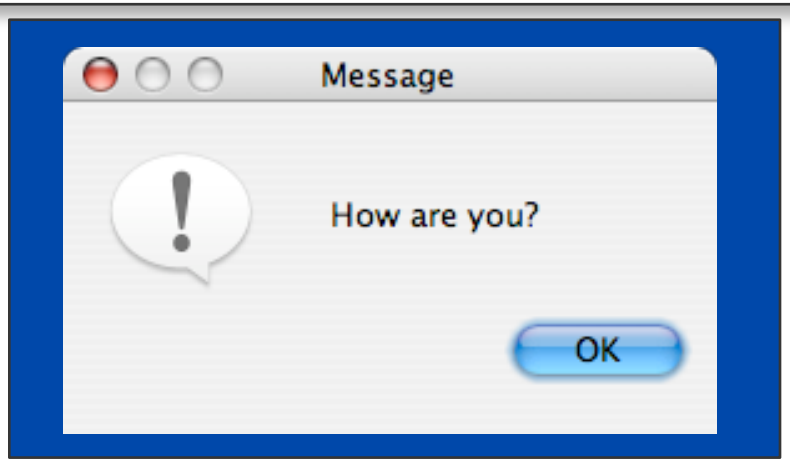


This output will appear at the console window.

JOptionPane

- Using **showMessageDialog** of the **JOptionPane** class is a simple way to bring up a window with a message.

```
JOptionPane.showMessageDialog(null, "How are you?");
```

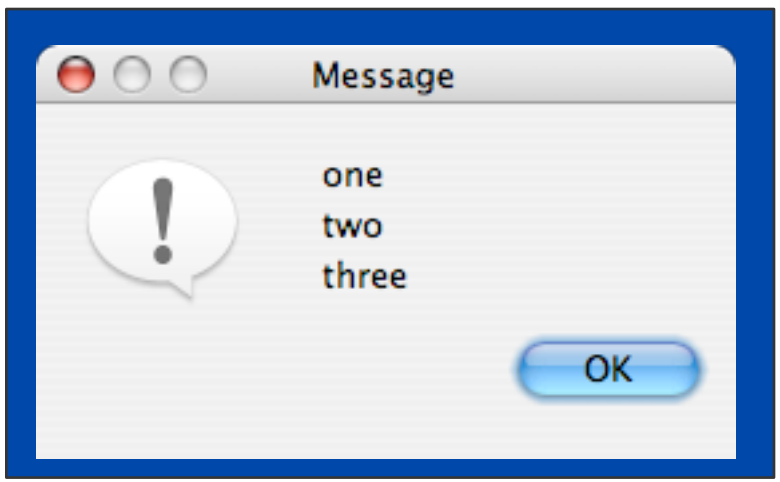


This dialog will appear at the center of the screen.

Displaying Multiple Lines of Text

- We can display multiple lines of text by separating lines with a new line marker \n.

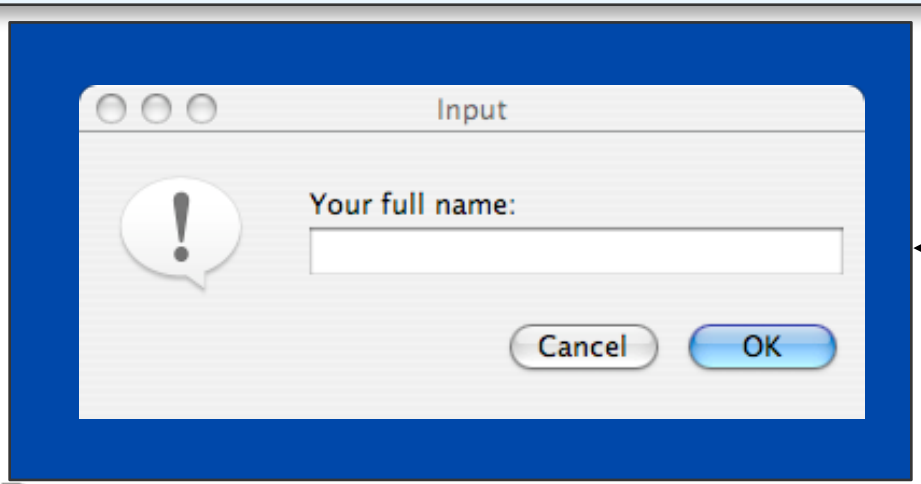
```
JOptionPane.showMessageDialog(null,“one\ntwo\nthree”);
```



JOptionPane for Input

- Using **showInputDialog** of the **JOptionPane** class is another way to input a string.

```
String name;

name = JOptionPane.showInputDialog
                    (null, "Your full name:");
```



This dialog will appear at the center of the screen ready to accept an input.

Standard Input and Scanner

- The System class has a special object that accepts input from the keyboard: System.in
- It reads only one byte at a time. We often need to read multiple bytes at a time.
- The Scanner class provides the necessary methods.
- A scanner object is created that “wraps” the System.in object.
- Calls to the method next() return one “word” at a time from the standard input
- Words are separated by whitespaces.

Standard Input and Scanner

```
import java.util.*;
...
Scanner scanner;
scanner = new Scanner(System.in);
System.out.print("Enter your first name: ");
System.out.println("Hello "+ scanner.next() +
".");
```

```
> Enter your first name: Lisa ↩
> Hello Lisa.
```

"Lisa" is typed by the user followed by the Enter (Return) key

Program Components

- A Java program is composed of
  - comments,
  - **import** statements, and
  - class declarations.

24

Sample Program

```
import javax.swing.*;
import java.util.*;

class SimpleProgram {

   public static void main(String[ ] args) {
      String name;
      name = JOptionPane.showInputDialog("Enter
         your name:");
      System.out.println("Hello " + name);
   }
}
```

Data and Identifiers

- Program = Data + Instructions
- Data is stored in memory.
- In Java, each piece of data has
  - a location in memory
  - an identifier (name)
  - a type
    - specifies legal values and operations (methods)
- Identifier names have rules and conventions (Recitation).
- Data are also called variables.

Data Types

- The primary type of data in OOP is objects.
- An object is composed of other object(s), and/or primitive data types.
- In contrast, objects are reference data types.
- There is no class for primitive data.
- There are three groups of primitive types:
  - Character: `char`
  - Numeric: `byte, short, int, long, float, double`
  - Boolean: `boolean`

Primitive Data Types

Characters

- In Java, single characters are represented using the data type **char**.
- Character values are written as symbols enclosed in single quotes.
- Characters are stored in memory using some form of encoding.
- *ASCII*, which stands for *American Standard Code for Information Interchange*, is one of the document coding schemes widely used today.
- Java uses Unicode, which includes ASCII, for representing **char** values.

Unicode Encoding

- The *Unicode Worldwide Character Standard* (*Unicode*) supports the interchange, processing, and display of the written texts of diverse languages.
- A UNICODE character takes up two bytes. ASCII characters take up one byte.

```
char ch1 = 'X';

System.out.println(ch1);                    X
System.out.println( (int) ch1);             88
```

Character Processing

```
char ch1, ch2 = 'X';
```

Declaration and initialization

```
System.out.print("ASCII code of character X is " +
                 (int) 'X' );

System.out.print("Character with ASCII code 88 is "
        + (char)88 );
```

Type conversion between int and char.

```
'A' < 'c'
```

This comparison returns true because ASCII value of 'A' is 65 while that of 'c' is 99.

```
if( ch1 < 'A' && ch2 == 99)
        System.out.print("Done");
```

Can compare characters with numbers directly.

Numeric Data Types

The various data types differ in the precision of the values they can hold.

Type	Content	Default Value	Size (bytes)	Minimum Value	Maximum Value
byte	Integer	0	1	-128	127
short			2	-32768	32767
int			4	-2147483648	2147483647
long			8	-9.22337E+18	9.22337E+18
float	Real	0.0	4	$-3.40282347 \times 10^{38}$	$3.40282347 \times 10^{38}$
double			8	-1.7977×10^{308}	1.7977×10^{308}

Arithmetic Operators

- The following table summarizes the arithmetic operators available in Java.

Operation	Java Operator	Example	Value (x=10, y=7, z=2.5)
Addition	+	x + y	17
Subtraction	-	x - y	3
Multiplication	*	x * y	70
Division	/	x / y x / z	1 4.0
Modulo division (remainder)	%	x % y	3

This is integer division where the fractional part is truncated.

Arithmetic expressions

- An arithmetic expression is composed of numeric values, numeric variables, and operators.
- For example, given: `int i,j;`

```
i + 3
(i + 2*(j-i))
-i + j
```

- Expressions can be used to assign values:

```
i = j + 3;
```

Take the value of j, add 3 to it and assign that value to i.

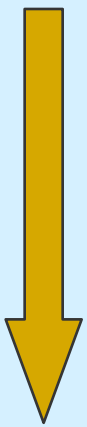
Order of evaluation

- How is the following expression evaluated?

  ```
  x + 3 * y
  ```

  Answer: `x` is added to `3*y` .
- We determine the order of evaluation by following *precedence rules*.
- Evaluation is in order of precedence.
- Operators at same level are evaluated *left to right* for most operators.

Precedence Rules

Order	Group	Operator	Rule
High	Subexpression	( )	Starting with innermost ()
	Unary operators	-, +	Left to right.
	Multiplicative operators	*, /, %	Left to right.
Low	Additive operators	+, -	Left to right.

Precedence Examples

```
x + 4*y - x/z + 2/x = ?
4   5   6
    1   2   3

x + (4*y) - (x/z) + (2/x)
```

```
(x + y * (4 - x) / z + 2 / -x ) =?
6     3     1   4   7   5   2

(x + ((y * (4-x)) / z) + (2 / (-x)))
```

To be safe, use parentheses!

Shorthand operators

- Some assignments and operators are combined into one operator to ease programming.

Operator	Usage	Meaning
+=	a+=b;	a=a+b;
-=	a-=b;	a=a-b;
*=	a*=b;	a=a*b;
/=	a/=b;	a=a/b;
%=	a%=b;	a=a%b;

Checkpoint

```
x = y*4-z/5
```

Which of these is equivalent to the above expression?

```
A. x = ((y*4)-z)/5
B. x = (y*(4-z))/ 5
C. x = (y*4)-(z/5)
D. x = y * ((4-z)/5)
```

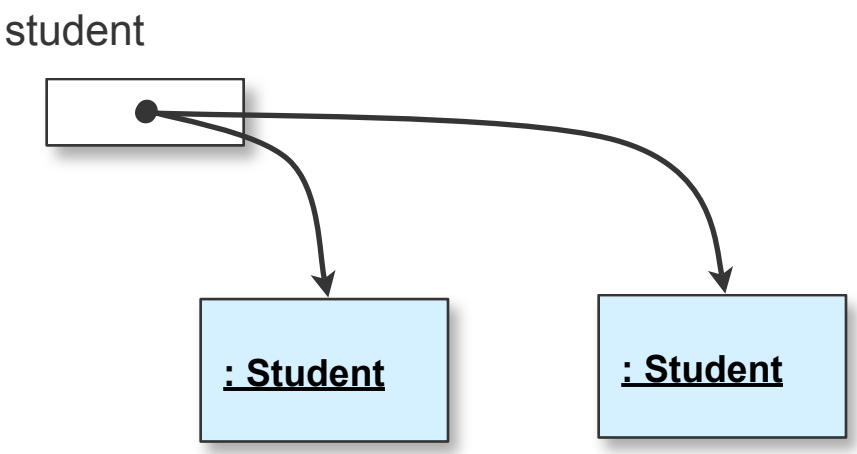
Reference Data Types

Objects

- Most data in OOP is organized as objects.
- An object is a collection of other objects and/or primitive data.
- Each object belongs to a Class.
- The class determines
  - the structure of each object (names and types of data members)
  - the behavior (allowed methods)

Object creation and assignment

```
Student   student;
student   =  new  Student( );
student   =  new  Student( );
```



The identifier student is allocated.

The **reference** to the first object is stored in student.

The **reference** to the second object is stored in student. The old reference is lost.

42

Assigning Primitive Data

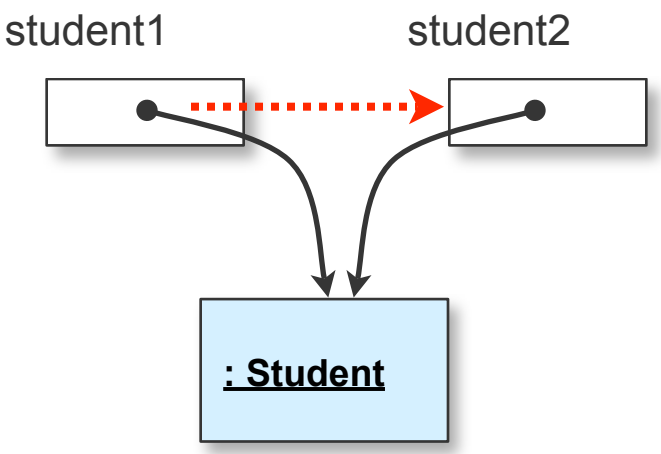
```
int   i,j;
i    =  5;
j    =  i;
```

Memory is allocated.

The **value** stored in i is copied to j.

Assigning objects

```
Student   student1, student2;
student1   =  new  Student( );
student2   =  student1;
```



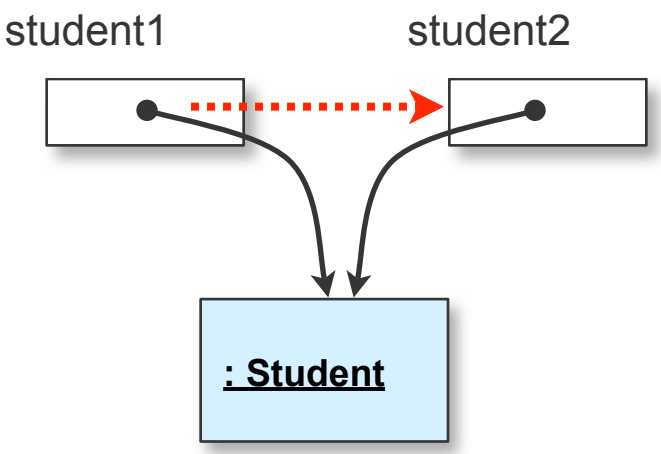
The identifiers are allocated.

The **reference** to the object is stored in student1.

The **reference** stored in student1. is copied to student2.

Really the same

```
Student   student1, student2;
student1   =  new  Student( );
student2   =  student1;
```



The **value** happens to be a reference to an object.

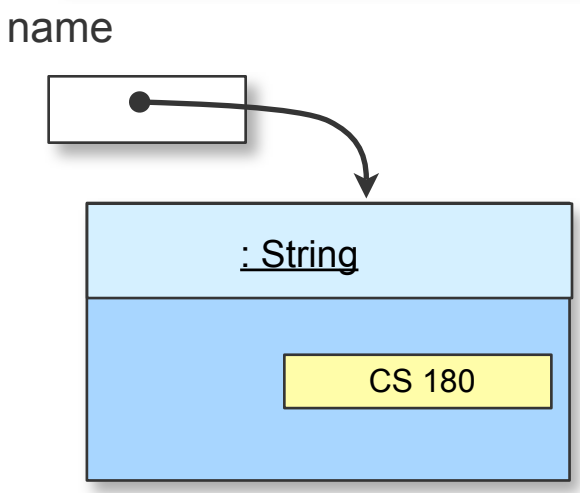
Hence **reference** type vs. **primitive** type.

The **value** stored in student1 is copied to student2.

Strings

Strings are reference types

```
1  String   name;
2  name  =  new  String("CS 180");
```

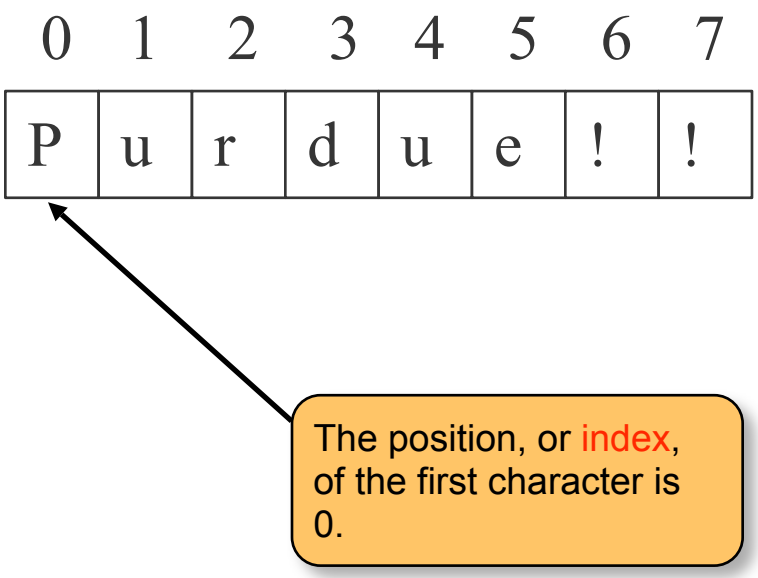


1. The identifier name is declared and space is allocated in memory.

2. A String object is created and the identifier name is set to refer to it.

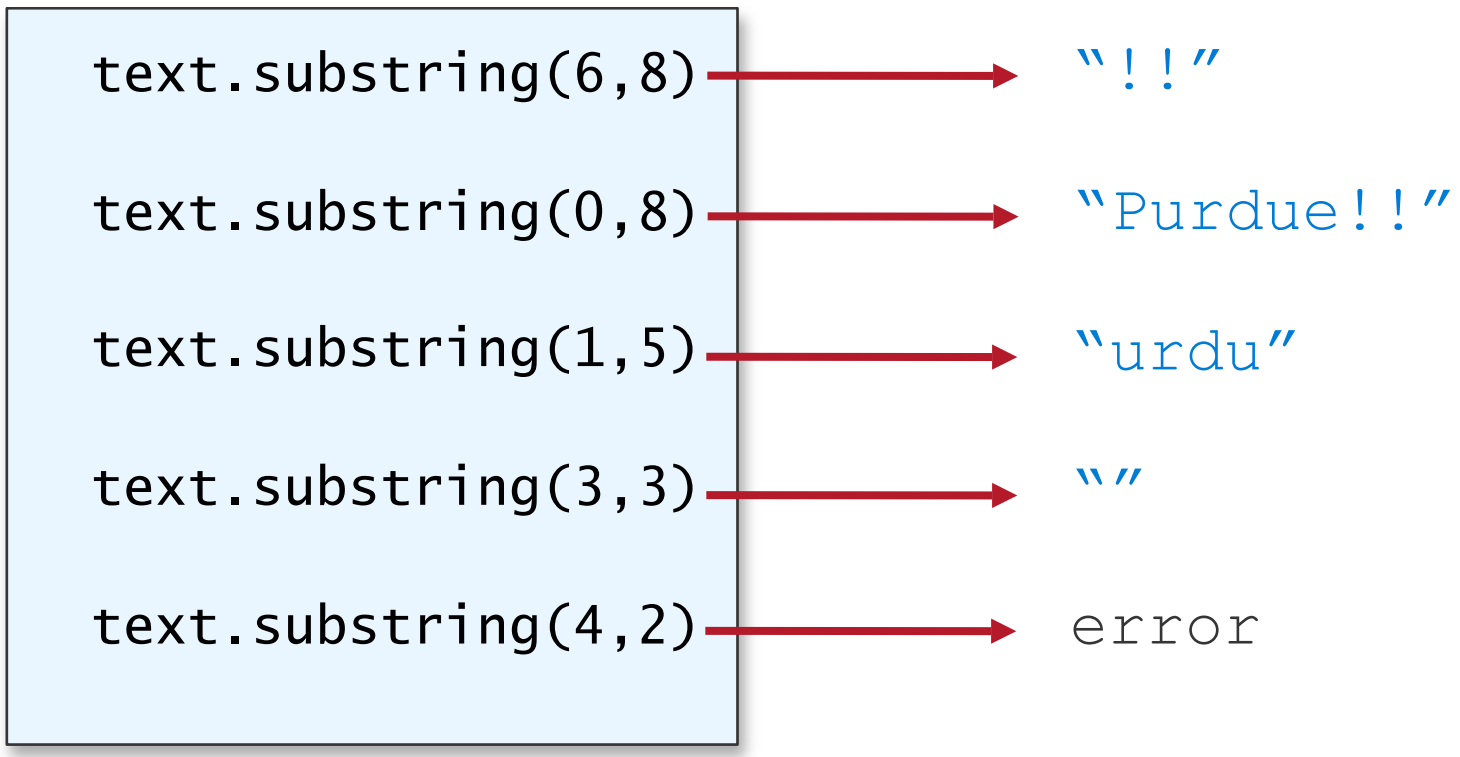
String Indexing

```
String text;
text = "Purdue!!";
```



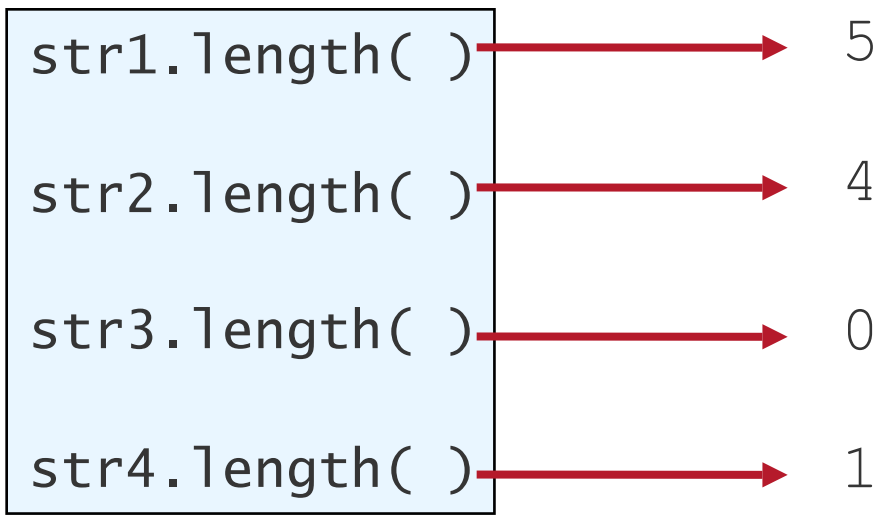
The substring method

```
String text = "Purdue!!";
```



The length method

```
String str1, str2, str3, str4;
str1 = "Hello" ;
str2 = "Java" ;
str3 = "" ; //empty string
str4 = " " ; //one space
```



The indexOf method

```
String str;
str = "I Love Java and Java loves me." ;
```


```
str.indexOf( "J" )
str.indexOf( "love" )
str.indexOf( "ove" )
str.indexOf( "Me" )
```

7

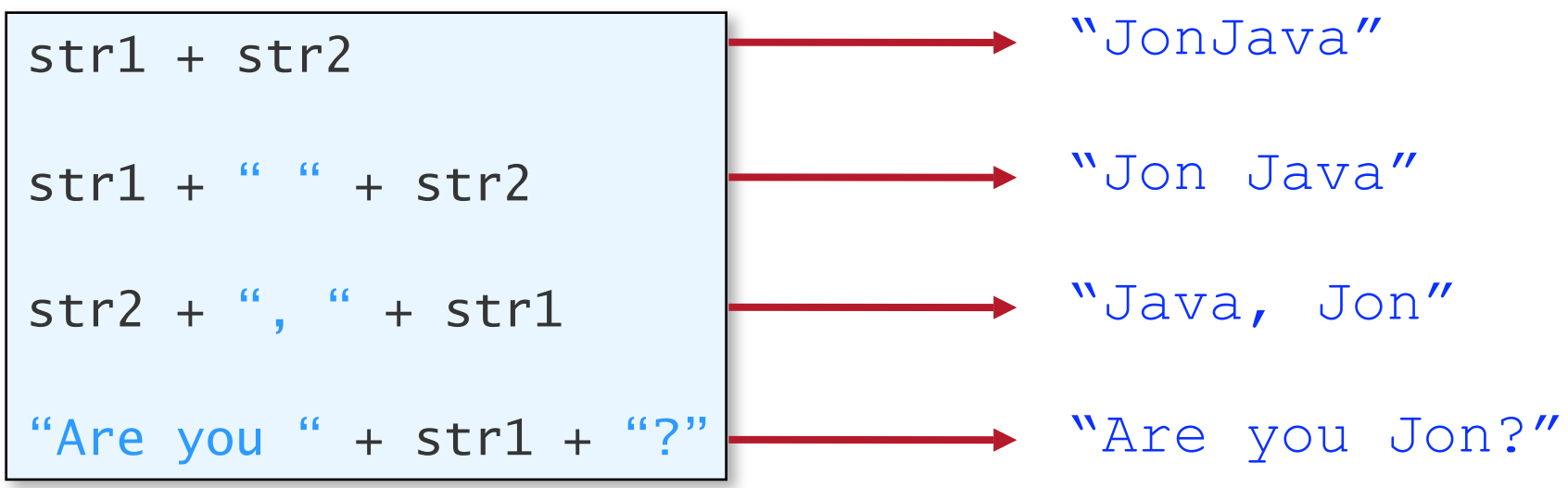
21

3

-1

The concatenation operator

```
String str1, str2;
str1 = "Jon" ;
str2 = "Java" ;
```



Checkpoint

```
String name;
name = “Cheong” ;
```

- What value does this method return?

```
name.length( );
```

Other Issues

- Type Specifics
- Constants
- Imprecision
- Math Class

Expression Types

- What is the data type of

```
i + j;
```

- Depends upon the types of `i` and `j`.
- If they are both
  - `int` then the result is also an `int`
  - `double` then the result is also a `double`
  - `long` … `long`
  - etc.
- Similarly for the other operators: -,*, …

Type Casting

- If **x** is a **float** and **y** is an **int**, what is the data type of

  x * y ?

  The answer is **float**.
- The above expression is called a *mixed expression*.
- Operands in mixed expressions are converted to a common type based on *promotion rules*.
- All are converted to the type with the highest precision in the expression.
- The entire expression is of this type too.

Implicit Type Casting

- Consider the following expression:

```
double x = 3 + 5;
```

- The result of 3 + 5 is of type **int**. However, since the variable **x** is **double**, the value 8 (type **int**) is promoted to 8.0 (type **double**) before being assigned to **x**.
- byte ➡ short ➡ int ➡ long ➡ float ➡ double
- Notice that it is a promotion. Demotion is not allowed.

```
int x = 3.5;
```

A higher precision value cannot be assigned to a lower precision variable.

Explicit Type Casting

- Instead of relying on the promotion rules, we can make an explicit type cast:

```
( <data type> ) <expression>
```

- Example

```
(float) x / 3
```

Type cast **x** to **float** and then divide it by 3.

```
(int) (x / y * 3.0)
```

Type cast the result of the expression **x / y * 3.0** to **int**.

- **NOTE**: Only the type of the return values is changed -- not the data itself.

Explicit demotion

- Promotion is automatically done whenever necessary.
- Demotion is not automatic, but can be forced:

```
int x;
double y;
y = 3.5;
x = (int)y;
```

- Assigning double (or float) to integer types results in *truncation* (not rounding).

Type Mismatch

- Suppose we want to input an age. Will this work?

```
int     age;

age = JOptionPane.showInputDialog(null, "Enter your age");
```

- No.
  A string value cannot be assigned directly to an int variable.

Type Conversion

- *Wrapper classes* are used to perform necessary type conversions, such as converting a String object to a numerical value.

```
int      age;
String   inputStr;

inputStr = JOptionPane.showInputDialog(
              null, "Enter your age");

age = Integer.parseInt(inputStr);
```

Other Conversion Methods

Class	Method	Example
Integer	parseInt	Integer.parseInt(“25”) → 25 Integer.parseInt(“25.3”) → error
Long	parseLong	Long.parseLong(“25”) → 25L Long.parseLong(“25.3”) → error
Float	parseFloat	Float.parseFloat(“25.3”) → 25.3F Float.parseFloat(“ab3”) → error
Double	parseDouble	Double.parseDouble(“25”) → 25.0 Integer.parseDouble(“ab3”) → error

Constants

- We can change the value of a variable. If we want the value to remain the same, we use a *constant*.

```
final double PI              = 3.14159;
final int    DAYS_IN_YEAR    = 365;
```

The reserved word **final** is used to declare constants.

These are constants, also called named constants.

These are called literal constants.

Why use Constants?

- Consistent value
  - No errors due to mistyping.
- Easy to manage
  - If we need to change the precision of PI, then we change it only once in the program.
- Programs are more readable.

CAUTION: Imprecision

- It is not possible to exactly represent every possible float (double) number
  - Fixed number of bits
    - Float: 4 bytes -- 32 bits:2^{32} (~1 billion) values
    - double: 8 bytes -- 64 bits: 2^{64} (~1 million trillion) values
  - Infinite numbers (e.g. between 1.0 and 2.0)!
- Floats and doubles may only store an approximation of the actual number!!!!
- Do not rely on exact values!
- Integers are stored precisely though!

The Math class

- The **Math** class in the **java.lang** package contains class methods for commonly used mathematical functions.

```
double      num, x, y;

x = …;
y = …;

num = Math.sqrt(Math.max(x, y) + 12.4);
```

Some Math Class Methods

Method	Input type	Output type	Description
exp(a)	double	double	Return *e* raised to power a.
log(a)	double	double	Return natural log of **a**.
floor(a)	double	double	Return largest whole number smaller than a.
max(a,b)	int double …	int double …	Return larger of a or b.
pow(a,b)	double	double	Return a raised to power b.
sqrt(a)	double	double	Return square root of a.
sin(a)	double	double	Return sine of a(in radians).

Control Flow

Flow of control

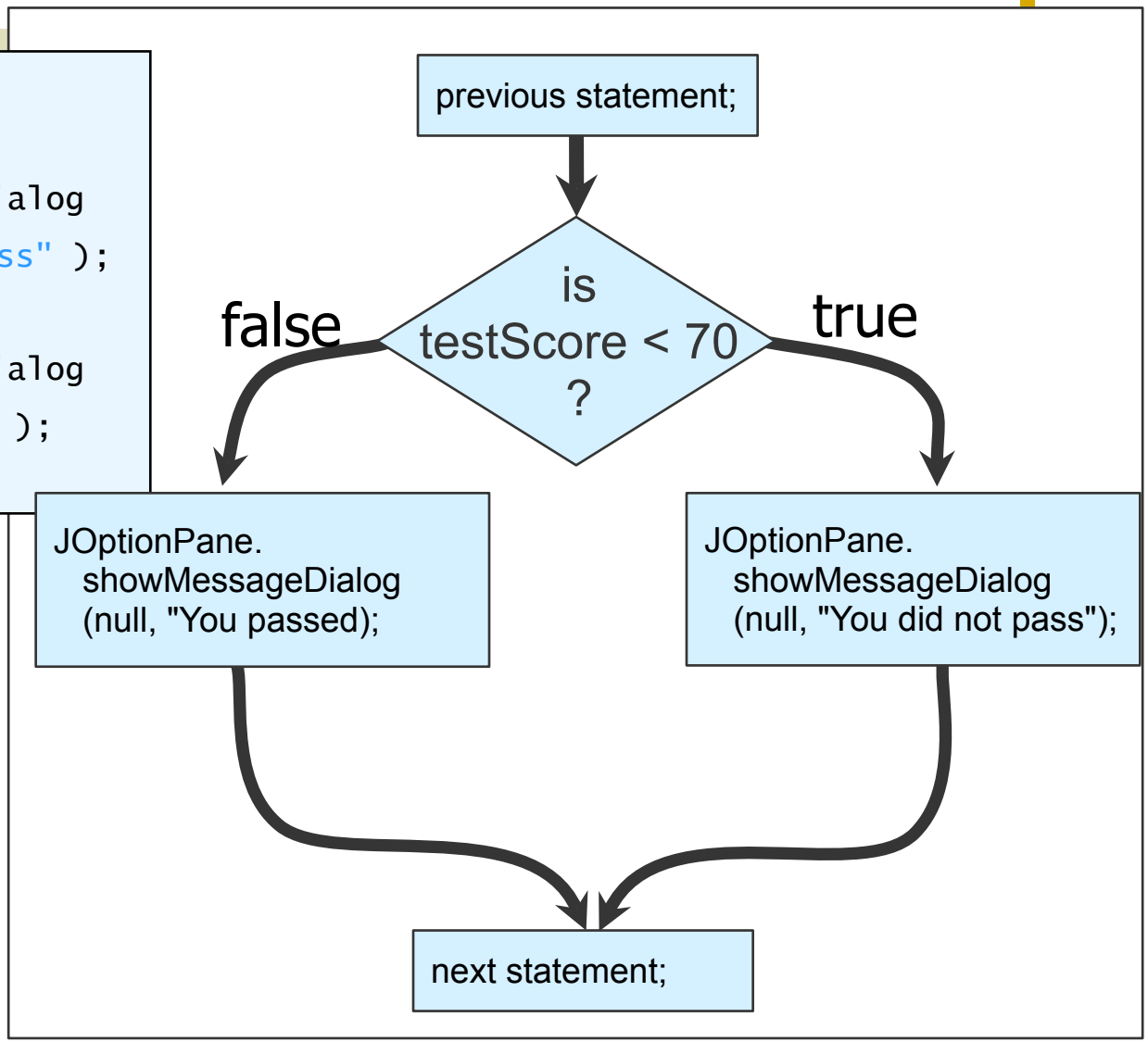
- Once a statement is executed, the next statement of the program is executed.
- Calling a method transfers the control to the statements in the method.
- Once the method returns, control returns to statement that made the call.
- Changing this flow of control is achieved using if and switch (and other) statements.
- These are called control flow statements.

Boolean expressions

- boolean is a primitive data type.
- A boolean expression can take only two values: **true** or **false**
- A simple boolean expression compares two values using a relational operator, e.g.
  - testScore < 70
  - j > i
  - balance == 100;
- Operators:
  - <, > , ==, !=, >=, <=

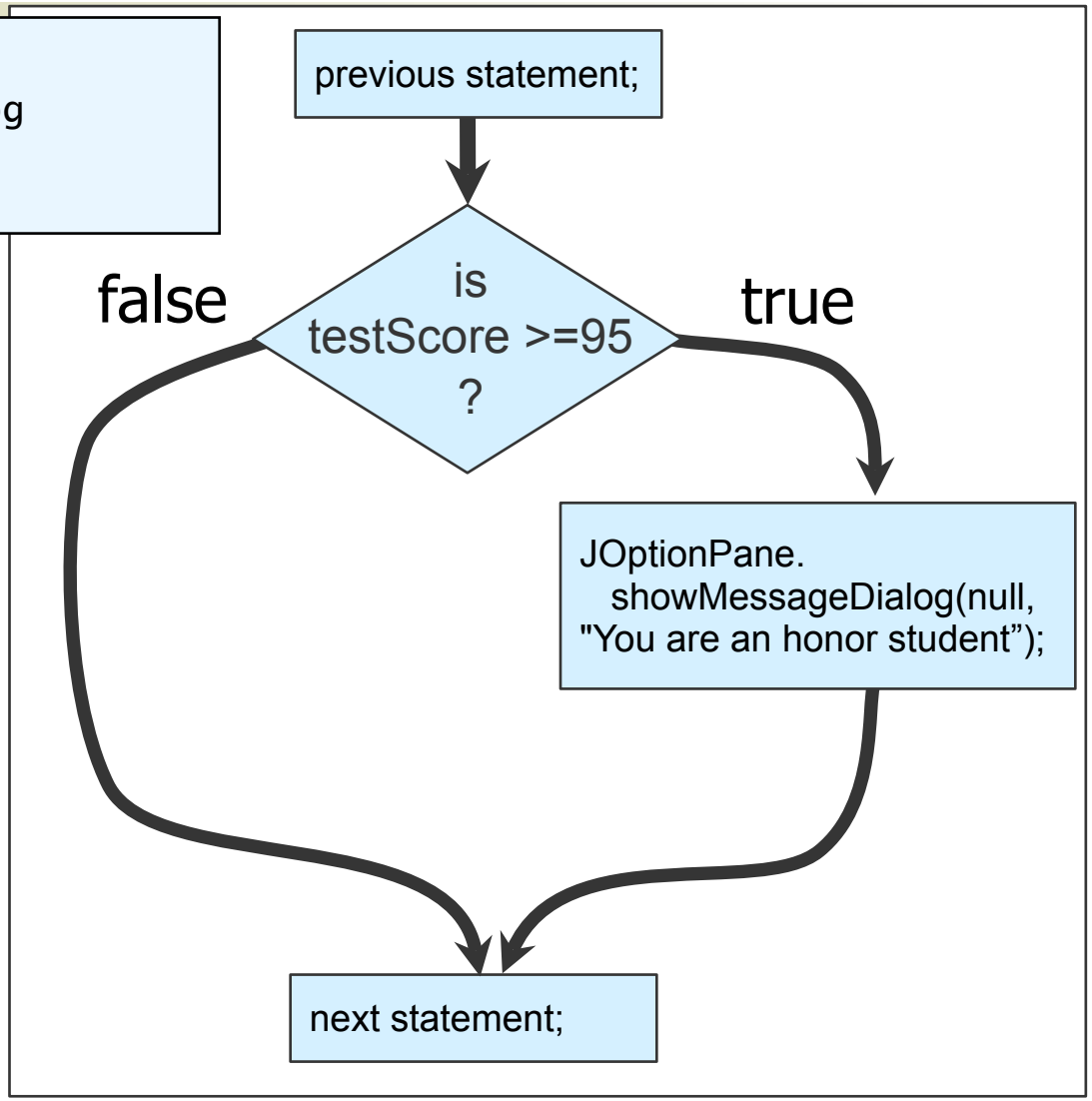
The if-then-else statement

```
<previous statement>
if (testScore < 70)
     JOptionPane.showMessageDialog
          (null,"You did not pass" );
else
     JOptionPane.showMessageDialog
           (null, "You passed " );
<next statement>
```



Control Flow with no else

```
if (testScore >= 95)
    JOptionPane.showMessageDialog
        (null, "You are an honor
student");
```



PURDUE UNIVERSITY

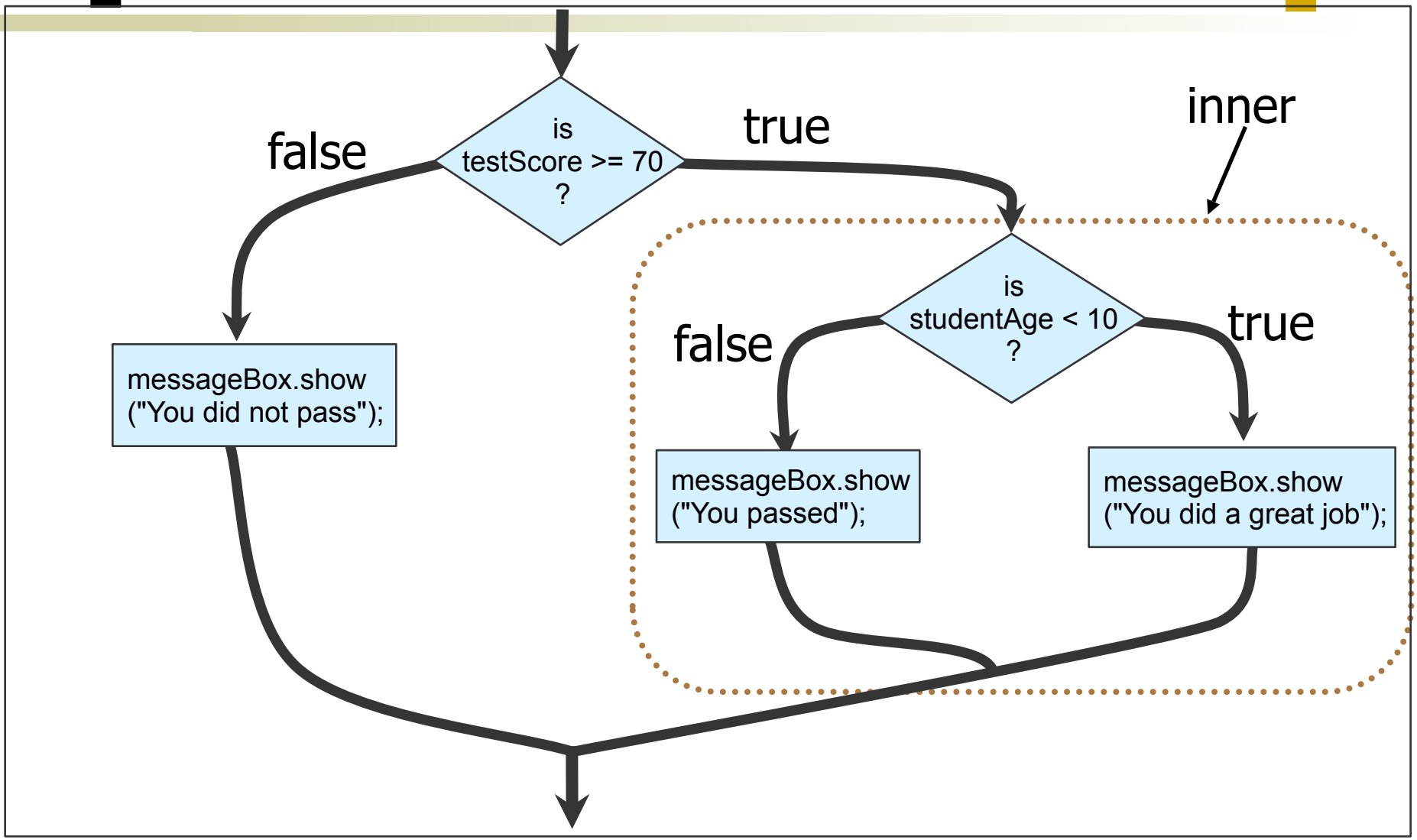
72

The Nested-if Statement

- The then and else block of an if statement can contain any valid statements, including other if statements. An if statement containing another if statement is called a nested-if statement.

```
if (testScore >= 70) {
    if (studentAge < 10) {
        System.out.println("You did a great job ");
    } else {
        System.out.println("You passed"); //test score >= 70
    }                                     //and age >= 10
} else { //test score < 70
     System.out.println("You did not pass");
}
```

Control Flow



Matching else

Are A and B different?

A

```
if (x < y)
    if (x < z)
        System.out.print("Hello");
else
    System.out.print("Good bye");
```

B

```
if (x < y)
    if (x < z)
      System.out.print("Hello");
    else
      System.out.print("Good bye");
```

Both A and B mean...

```
if (x < y) {
    if (x < z) {
        System.out.print("Hello");
    } else {
        System.out.print("Good bye");
    }
}
```

Each else paired with nearest unmatched if -- use braces to change this as needed.

The switch statement

- The if statement is essential for writing interesting programs.
- Other control flow statements (e.g., switch and loops) can be implemented using if statements.
- They are available since we often need them. Programs are more readable too.
- Next: switch

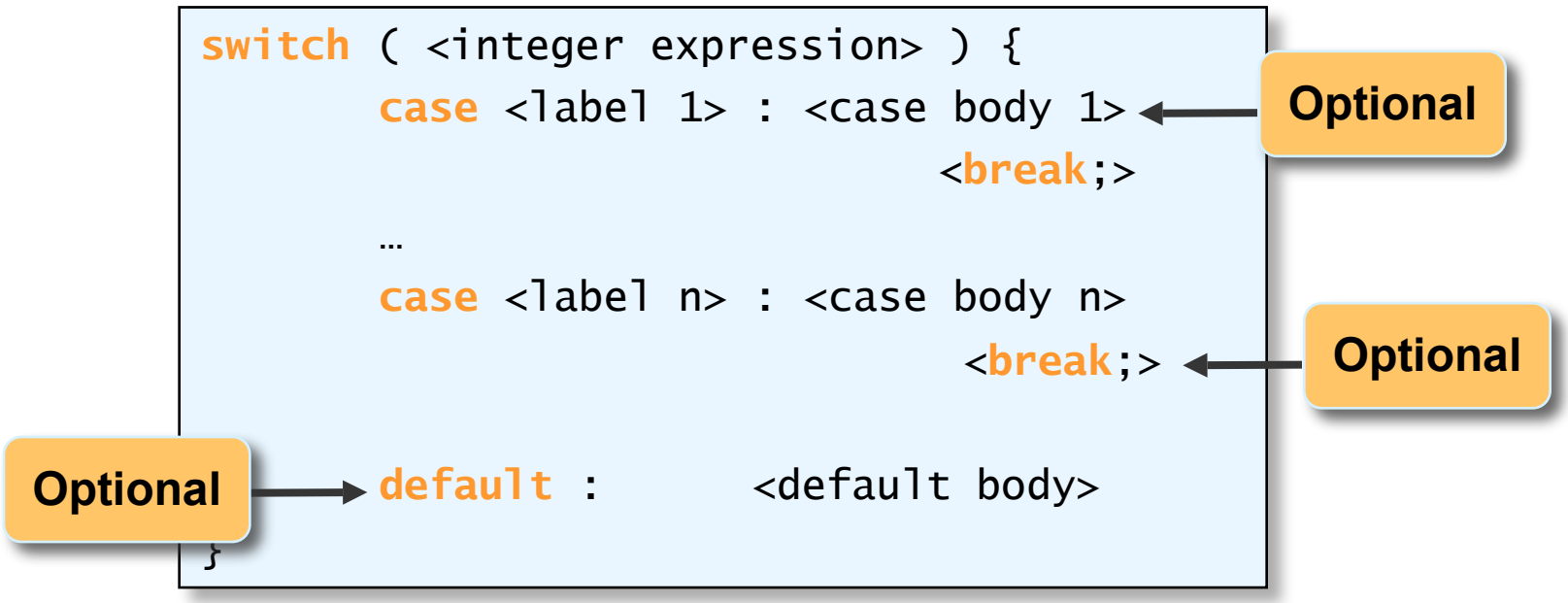
The switch Statement

```
int recSection;
recSection =
Integer.parseInt(JOptionPane.showInputDialog("Recitation Section
(1,2,…,4):" ));

switch (recSection) {

    case 1: System.out.print("Go to UNIV 101");
            break;

    case 2: System.out.print("Go to UNIV 119");
            break;

    case 3: System.out.print("Go to STON 217");
            break;

    case 4: System.out.print("Go to UNIV 101");
            break;
}
```

This statement is executed if the gradeLevel is equal to 1.

This statement is executed if the gradeLevel is equal to 4.

Syntax for the switch Statement



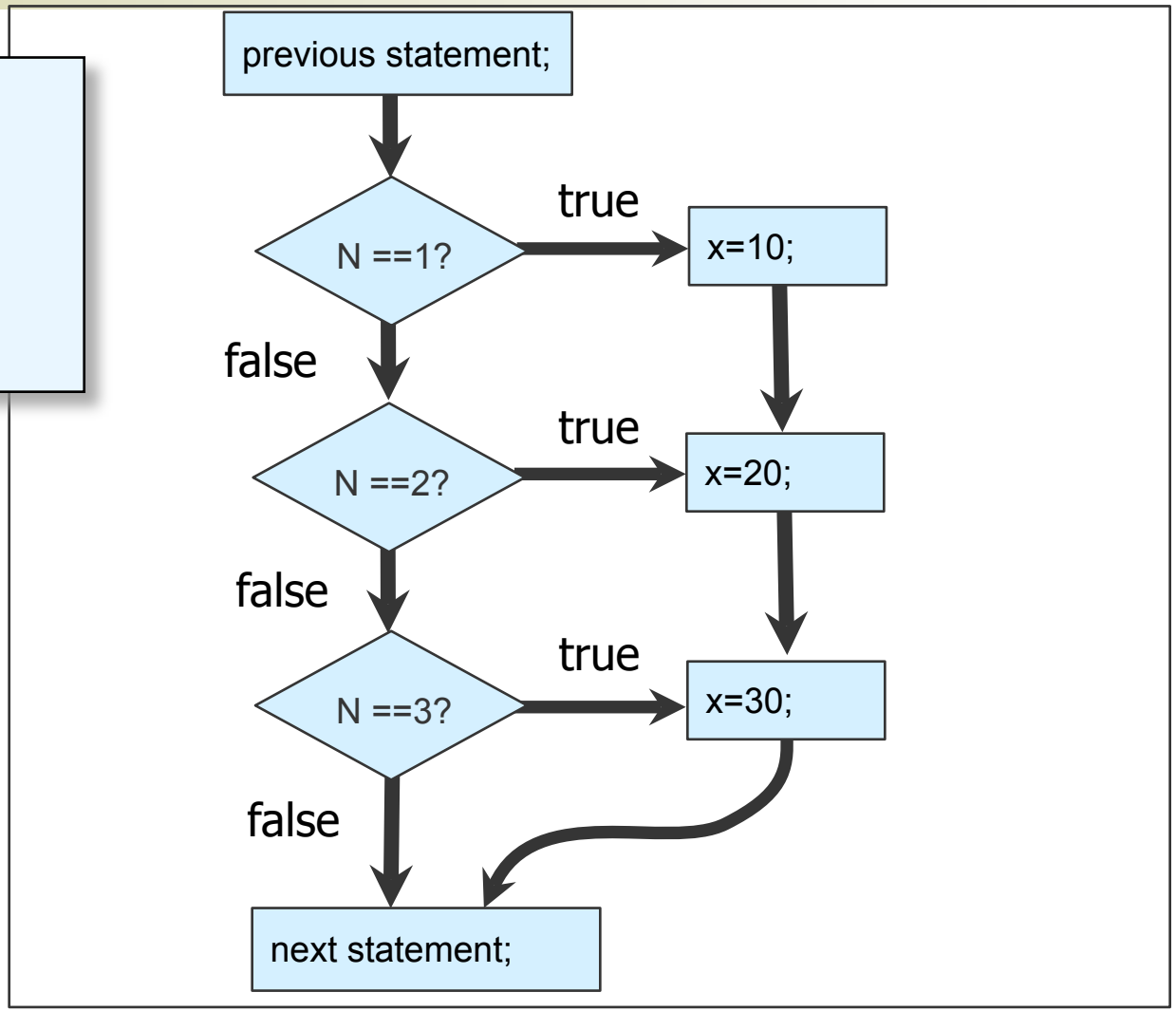
The break statement is *optional* within each case.
A case body is also *optional*.
The default is *optional* for the switch statement.

Switch statement (cont.)

- The integer expression can have only one of the following types:
  - char, byte, short, or int
- The label must be a literal or named constant of the same type.
- Each case body *may* end with a break statement.
- A break causes the execution to go to the statement following the switch statement.
- The default case applies when no label matches.
- Each label must be unique.
- Labels may be listed in any order.

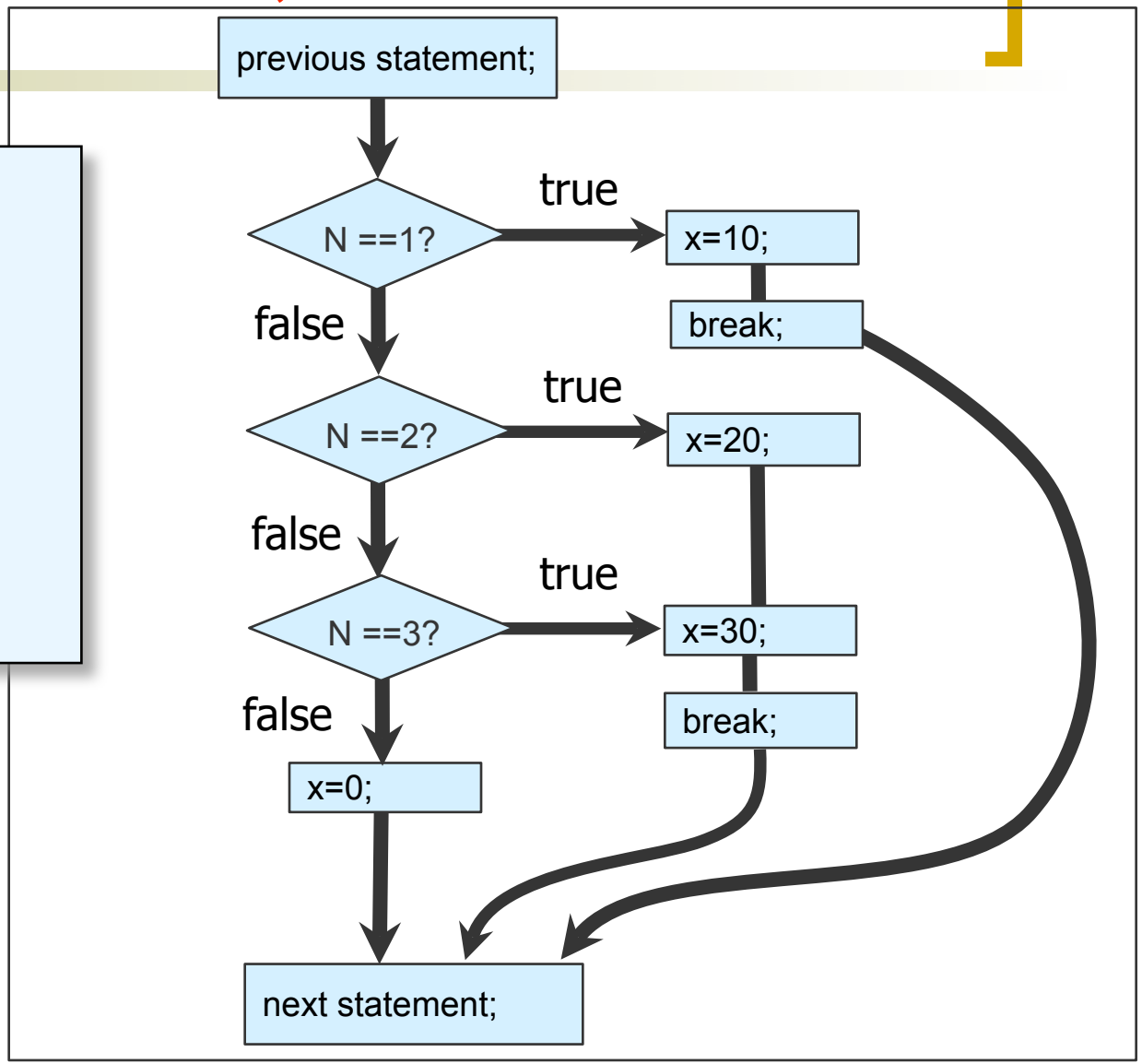
Simple switch statement

```
switch ( N ) {
    case 1: x = 10;
    case 2: x = 20;
    case 3: x = 30;
}
```



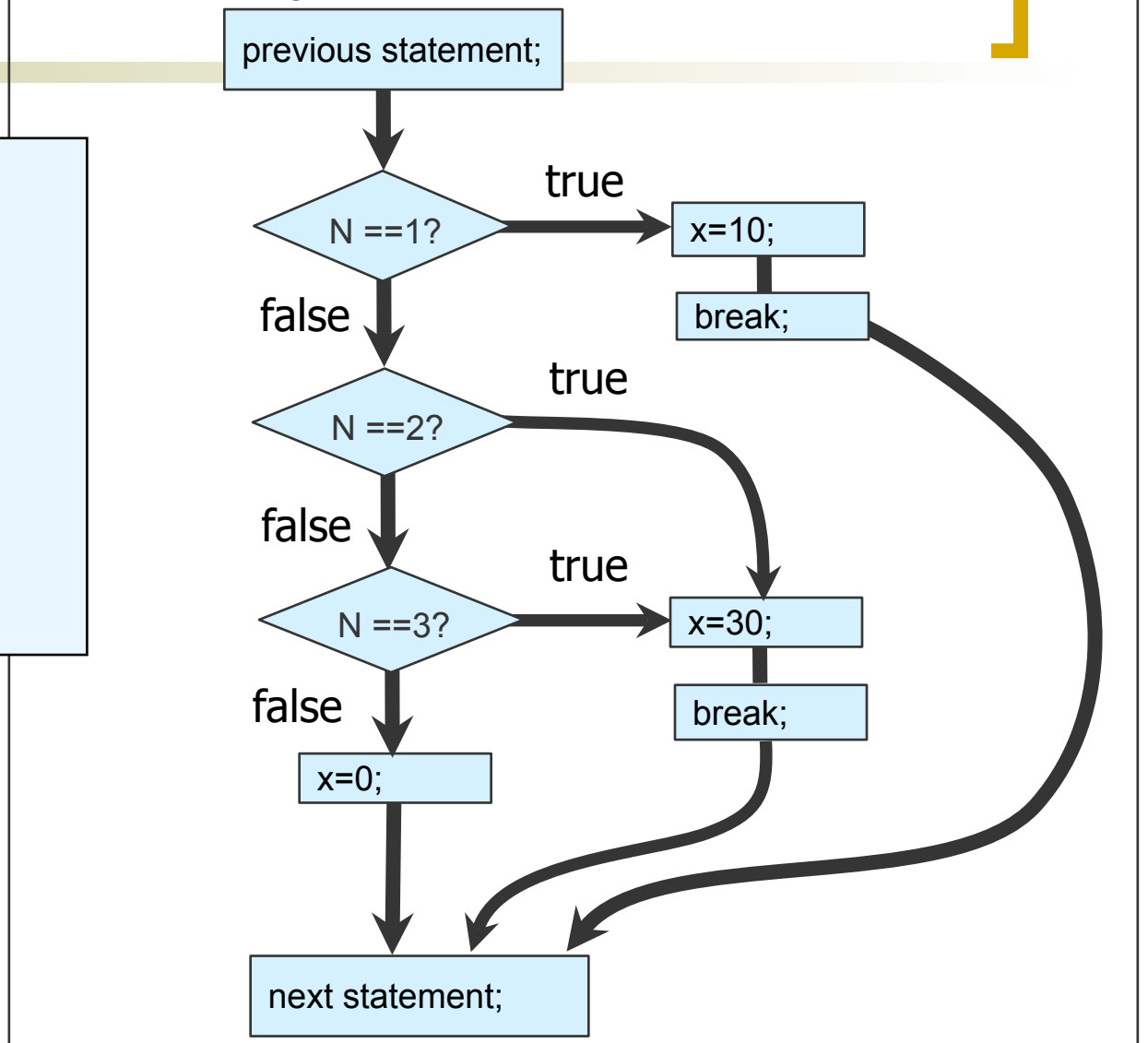
switch with break, and default

```
switch ( N ) {
    case 1: x = 10;
            break;
    case 2: x = 20;
    case 3: x = 30;
            break;
    default: x = 0;
}
```



Missing case body

```
switch ( N ) {
    case 1: x = 10;
            break;
    case 2:
    case 3: x = 30;
            break;
    default: x = 0;
}
```



Checkpoint

- What is the output of the following code?

```
int x=3, y=2, z=1;
if (x < y)
   if (x < z)
       System.out.print("Inky");
else
   System.out.print("Pinky");
   System.out.print("Ponky");
```

A. Inky
B. Pinky
C. Ponky
D. PinkyPonky
E. No output

Boolean Operators

- Boolean expressions can be combined using boolean operators.
- A *boolean operator* takes boolean values as its operands and returns a boolean value.
- The boolean operators are
  - and &&
  - or ||
  - not !
  - exclusive-OR ^

Semantics of Boolean Operators

- Truth table for boolean operators

p	q	p && q	p || q	!p	p^q
false	false	false	false	true	false
false	true	false	true	true	true
true	false	false	true	false	true
true	true	true	true	false	false

- Sometimes true and false are represented by 1 and 0 (NOT in Java).
- In C and C++, 0 is **false,** everything else is **true**.

Short-Circuit Evaluation

- Consider the following boolean expression:

```
x > y || x > z
```

- The expression is evaluated left to right. If x > y is true, then there's no need to evaluate x > z because the whole expression will be true whether x > z is true or not.
- To stop the evaluation once the result of the whole expression is known is called *short-circuit evaluation*.
- What would happen if the short-circuit evaluation is not done for the following expression?

```
z == 0 || x / z > 20
```

Short-circuit evaluation

- Sometimes this is useful
  - it is more efficient
  - `z == 0 || x / z > 20`
- Can force complete evaluation by using:
  - & instead of &&
  - | instead of ||
- Short-circuit evaluation is also called *lazy evaluation* (as opposed to *eager evaluation*)
- NOTE: &, | also denote bitwise and and or

Operator Precedence Rules

Group	Operator	Precedence	Associativity
Subexpresion	( )	10 (Innermost first)	Left to Right
Postfix increment and decrement operators	++	9	**Right to Left**
	--		
Unary operators Prefix inc, decr	++	8	**Right to Left**
	--		
	-		
	!		
Multiplicative operators	*	7	Left to Right
	/		
	%		
Additive operators	+	6	Left to Right
	-		
Relational operators	<	5	Left to Right
	<=		
	>		
	>=		
Equality operators	==	4	Left to Right
	!=		
Boolean AND	&&	3	Left to Right
Boolean OR	||	2	
Assignment	=	1	**Right to Left**

Precedence Examples

```
int x= 1, y=10, z=100;
boolean bool, test=false;
```

x = -y + y * z;	x = (-y) + (y*z);
x ==1 && y > 5	(x ==1) && (y > 5)
4 < x && !test	(4<x) && (!test)
bool = x!=y && y == z	bool = (x!=y) && (y ==z)
x==y || y>4 && z<2	(x==y) || ((y>4) && (z<2))

Prefix operators

- The increment (++) and decrement (--) operators can precede the operand
  - x++; ++x; y--; --y;
- Their effect on the operand is the same, however, they vary only in terms of the timing of the increment or decrement.
- The postfix operators are applied AFTER the variable's value is used.
- The prefix operator are applied BEFORE

Example

```
int x=2, y=10;

x = y++;
System.out.println("X is:" + x);
System.out.println("Y is:" + y);
```

```
X is: 10
Y is: 11
```

```
int x=2, y=10;

x = y--;
System.out.println("X is:" + x);
System.out.println("Y is:" + y);
```

```
X is: 10
Y is: 9
```

```
int x=2, y=10;

x = ++y;
System.out.println("X is:" + x);
System.out.println("Y is:" + y);
```

```
X is: 11
Y is: 11
```

```
int x=2, y=10;

x = --y;
System.out.println("X is:" + x);
System.out.println("Y is:" + y);
```

```
X is: 9
Y is: 9
```

```
int x=2, y=10, z;

z = x++ * --y;
System.out.println("X is:" + x);
System.out.println("Y is:" + y);
System.out.println("Z is:" + z);
```

```
X is: 3
Y is: 9
Z is: 18
```

```
int x=2, y=10;

x = --x * ++y;
System.out.println("X is:" + x);
System.out.println("Y is:" + y);
```

```
X is: 11
Y is: 11
```

PURDUE UNIVERSITY

Side effects -- 1

```
int x= 1, y=10, z=100;
boolean bool, test=false;
```

- x = y++; x: 10 y: 11
- x = ++y; x: 11 y: 11
- x = -++y; x: -11 y: 11
- x = -y++; x: -10 y: 11
- x = -y--; x: -10 y: 9
- x = -(--y); x: -9 y: 9
- x = ++y++; ERROR!

Prefix vs. postfix.

- A prefix (postfix) operator is equivalent to executing the operator before (after) using the value of the variable:

```
z = x++ * --y;
```

- Is equivalent to:

```
y = y-1;
z = x * y;
x = x + 1;
```

What about:

```
z = x++ * x++;
```

PURDUE UNIVERSITY

More Examples

```
z = x++ * x++;
```

- Is equivalent to:

```
z = x * (x+1);
x = x+2;
```

```
x = x++ * --y;
```

- Is equivalent to:

```
y = y - 1;
x = x * (y-1) + 1;
```

Side effects -- 2

```
int x= 1, y=10, z=100;
boolean bool, test=false;
```

x = y = z;	x: 100 y: 100 z: 100
x = y = ++z;	x: 101 y: 101 z: 101
bool = (x=11)>y	x: 11 y: 10 bool: true
bool = (x=11)>y++	x: 11 y: 11 bool: true
bool = (x=11)> ++y	x: 11 y: 11 bool: false
(x=3) > y && (z=5)<10	x: 3 y: 10 z: 100
(x=3) > y & (z=5)<10	x: 3 y: 10 z: 5

Comparing Objects

- There is only one way to compare primitive data types, but with objects (reference data types) there are two:
  1. We can test whether two variables point to the same object (use ==), or
  2. We can test whether two distinct objects have the same contents.

Using == With Objects (Sample 1)

```
String str1 = new String("Java");
String str2 = new String("Java");

if (str1 == str2) {
     System.out.println("Equal");
} else {
     System.out.println("Not equal");
}
```

Not equal because str1 and str2 point to different String objects.

Using == With Objects (Sample 2)

```
String str1 = new String("Java");
String str2 = str1;

if (str1 == str2) {
     System.out.println("Equal");
} else {
     System.out.println("Not equal");
}
```

They are equal here because str1 and str2 point to the same object.

Using equals with String

```
String str1 = "Java";
String str2 = "Java";

if (str1.equals(str2)) {
     System.out.println("Equal");
} else {
     System.out.println("Not equal");
}
```

It's equal here because str1 and str2 have the same sequence of characters.

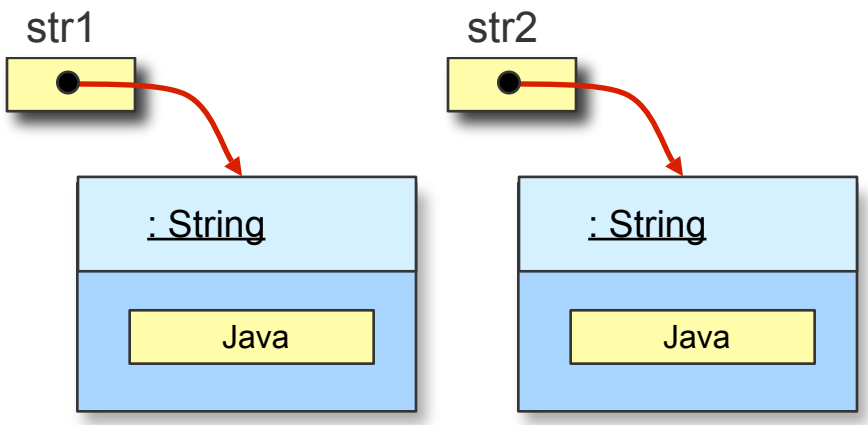
The Semantics of ==

Case 1: different objects

```
String str1, str2;

str1 = new String("Java");
str2 = new String("Java");
```

```
str1==str2 ?
```

false

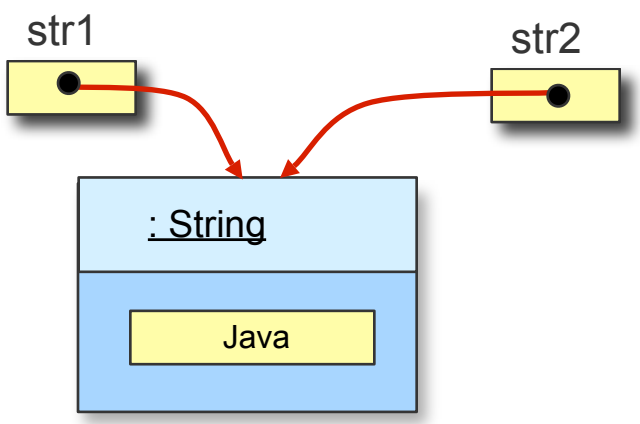


Case 2: same object

```
String str1, str2;

str1 = new String("Java");
str2 = str1;
```

```
str1==str2 ?
```

true



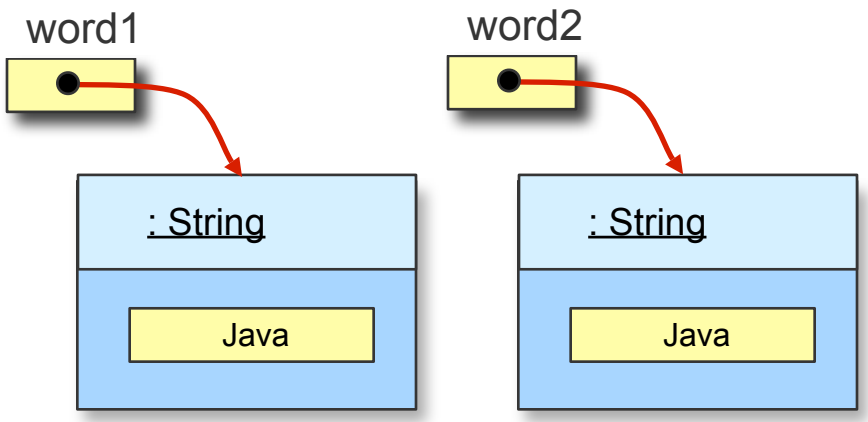
In creating String objects

```
String word1, word2;

word1 = new String("Java");
word2 = new String("Java");
```

```
word1==word2 ?
```

false

Whenever the **new** operator is used, there will be a new object.

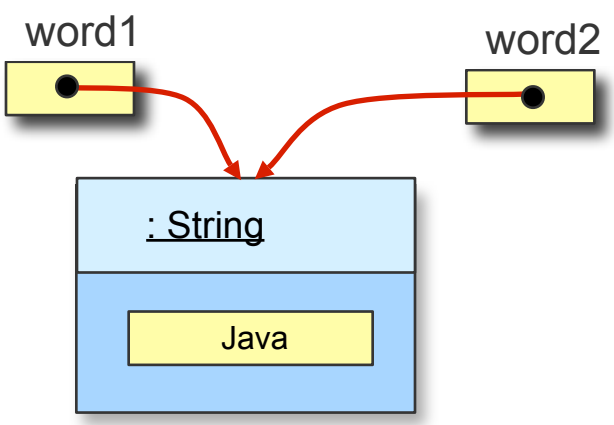


```
String word1, word2;

word1 = "Java";
word2 = "Java";
```

```
word1==word2 ?
```

true

Literal String objects such as "Java" will always refer to the same object.



Miscellaneous Recap

Identifiers

- In order to manipulate an object, we have to give it a name and also create the object.
- Names are also called **identifiers**
- An identifier
  - Cannot be a reserved word
  - Can consist only of letters(A..Z,a..z), digits(0..9), $ and _
  - Cannot begin with a digit
- These are required rules. We also have naming conventions that make programs easier to read
  - Identifiers begin with a lowercase letter
  - Class names begin with an uppercase letter
  - Camel case studentName

Source Code

- A program written in a machine language is called an executable, or a binary.
  - It is not portable.
- A program written in a HLL is often called source code.
- Given an executable, it is difficult to recover the source code (not impossible).
- Thus, companies release only the executables.
- This makes it hard for someone else to replicate the software and also to modify it (maybe even to trust it completely)
- Open-Source is an alternative approach.

Comparing Strings

- If we want to compare the contents of string objects, we can use *equals*

```
String word1, word2;
if(word1.equals(word2)){
      System.out.print("They are equal");
} else {
      System.out.print("They are not equal");
}
```

- There is also *equalsIgnoreCase* and *compareTo*
- equalsIgnoreCase treats upper and lower case letters as the same (e.g. 'H' and 'h')

compareTo method

- This method compares two strings in terms of their lexicographic order.

```
str1.compareTo(str2)
```

- It returns:
  - 0 if the strings are exactly the same;
  - a negative value if str1 comes before str2;
  - a positive value if str1 comes after str2;
- Lexicographic ordering is determined by UNICODE values.
  - …,!,…,+,-,… 0,1,2,…,9,…A,B,…,Z,…,a,b,…,z, …

Comparing Objects

- The operators <, >=, … cannot be applied to compare objects.
- In order to compare objects, we need to implement an appropriate method.
- For example, the *equals*, *compareTo* methods for strings.
- A default *equals* method exists for each class, but it may not behave as you expect.

Overloaded Operator +

- The plus operator + can mean two different operations, depending on the context.
- <val1> + <val2> is an addition if both are numbers. If either one of them is a String, then it is a concatenation.
- Evaluation goes from left to right.

output = “A” + 1 + 2;

output is “A12”

output = 1 + 2 + “A”;

output is “3A”

The DecimalFormat Class

- Use a DecimalFormat object to format the numerical output.

```
double num = 123.45789345;

DecimalFormat df = new DecimalFormat("0.000");
                  //three decimal places
```

```
System.out.print(num);
```

→ 123.45789345

```
System.out.print(df.format(num));
```

→ 123.458

The GregorianCalendar Class

- Use a GregorianCalendar object to manipulate calendar information

```
GregorianCalendar today, independenceDay;

today     = new GregorianCalendar();

independenceDay
          = new GregorianCalendar(1776, 6, 4);
             //month 6 means July; 0 means January
```

Retrieving Calendar Information

- This table shows the class constants for retrieving different pieces of calendar information from Date.

Constant	Description
YEAR	The year portion of the calendar date
MONTH	The month portion of the calendar date
DATE	The day of the month
DAY_OF_MONTH	Same as DATE
DAY_OF_YEAR	The day number within the year
DAY_OF_MONTH	The day number within the month
DAY_OF_WEEK	The day number within the week (Sun --1, Mon -- 2, etc.)
WEEK_OF_YEAR	The week number within the year
WEEK_OF_MONTH	The week number within the month
AM_PM	The indicator for AM or PM (AM -- 0, PM -- 1)
HOUR	The hour in the 12-hour notation
HOUR_OF_DAY	The hour in 24-hour notation
MINUTE	The minute within the hour

Sample Calendar Retrieval

```
GregorianCalendar cal = new GregorianCalendar();
            //Assume today is Nov 9, 2003

System.out.print("Today is " +
      (cal.get(Calendar.MONTH)+1) + "/" +
      cal.get(Calendar.DATE) + "/" +
      cal.get(Calendar.YEAR));
```

Output

```
Today is 11/9/2003
```

Clarification on Division

- Integer division yields a truncated integer answer.
  - 14/3 = 4
  - 14/-3 = -4
  - -14/3 = -4
  - -14/-3 = 4

Clarification on Modulo

- Modulo's sign matches the dividend's sign.
- Modulo satisfies the following:
  - a = ((a/b)*b) + (a%b)
- For example:
  - 14%3 = 2
  - 14%-3 = 2
  - -14%3 = -2
  - -14%-3 = -2