INTRODUCTION TO JAVA

- Java is related to C++, which is a direct descendent of C
- From C, Java derives its syntax
- Many of Java's object-oriented features were influenced by C++
- Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991

- This language was initially called “Oak” but was renamed “Java” in 1995
- Java can be used to create two types of programs: ***applications*** and ***applets***
- An *application* is a program that runs on your computer, under the operating system of that computer
- An *applet* is an application designed to be transmitted over the Internet and executed by a Java-compatible Web browser

- An applet is actually a tiny Java program, dynamically downloaded across the network, just like an image, sound file, or video clip

- An applet is a program that can react to user input and dynamically change—not just run the same animation or sound over and over

Security

- When you use a Java-compatible Web browser, you can safely download Java applets without fear of viral infection or malicious intent

- Java achieves this protection by confining a Java program to the Java execution access to other parts of the computer environment and not allowing it

- Java is portable across many types of computers and operating systems that are in use throughout the world

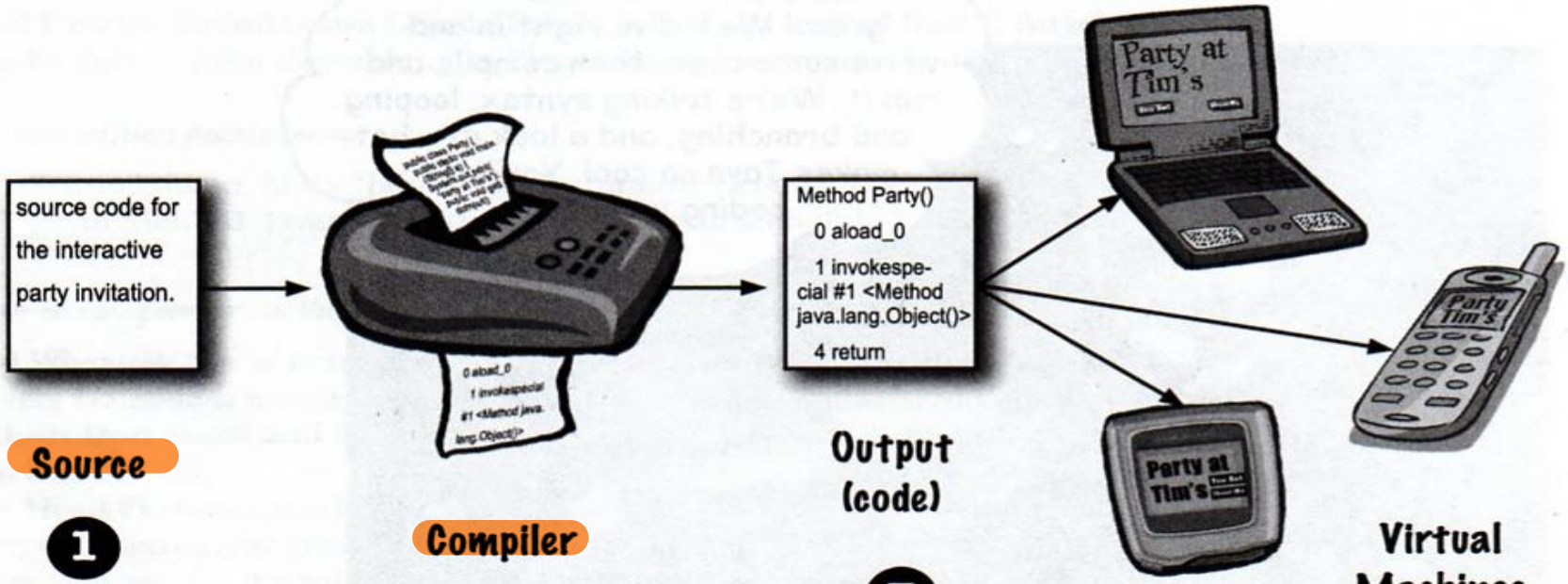
Java's Magic: The Bytecode

- The output of a Java compiler is not executable code ; rather, it is ***bytecode***

- *Bytecode* is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the ***Java Virtual Machine*** (*JVM*)

The Way Java Works

The goal is to write one application (in this example, an interactive party invitation) and have it work on whatever device your friends have.



What you'll do in Java

You'll type a source code file, compile it using the javac compiler, then run the compiled bytecode on a Java virtual machine.

```
import java.awt.*;
import java.awt.event.*;
class Party {
  public void buildInvite() {
    Frame f = new Frame();
    Label l = new Label("Party at Tim's");
    Button b = new Button("You bet");
    Button c = new Button("Shoot me");
    Panel p = new Panel();
    p.add(l);
  } // more code here...
}
```

Source

1

Type your source code.

Save as: ***Party.java***

```
File Edit Window Help Plead
%javac Party.java
```

Compiler

2

Compile the ***Party.java*** file by running `javac` (the compiler application). If you don't have errors, you'll get a second document named ***Party.class***

The compiler-generated Party.class file is made up of *bytecodes*.

```
Method Party()
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 return
Method void buildInvite()
  0 new #2 <Class java.awt.Frame>
  3 dup
  4 invokespecial #3 <Method java.awt.Frame()>
```

Output (code)

3

Compiled code: ***Party.class***

```
File Edit Window Help Swear
%java Party
```

Virtual Machines

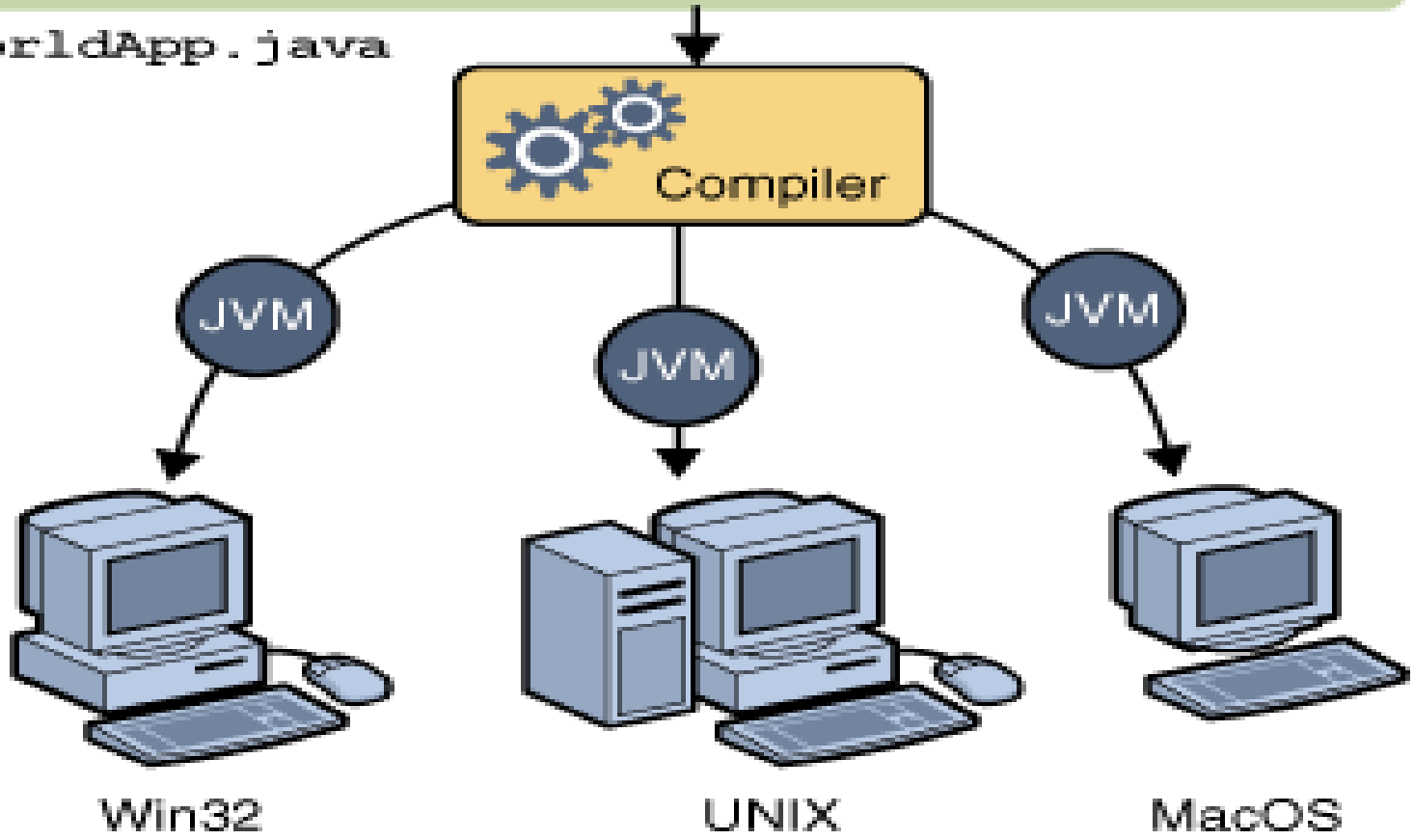
4

Run the program by starting the Java Virtual Machine (JVM) with the ***Party.class*** file. The JVM translates the *bytecode* into something the underlying platform understands, and runs your program.

Java Program

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

HelloWorldApp.java



Writing a class with a main

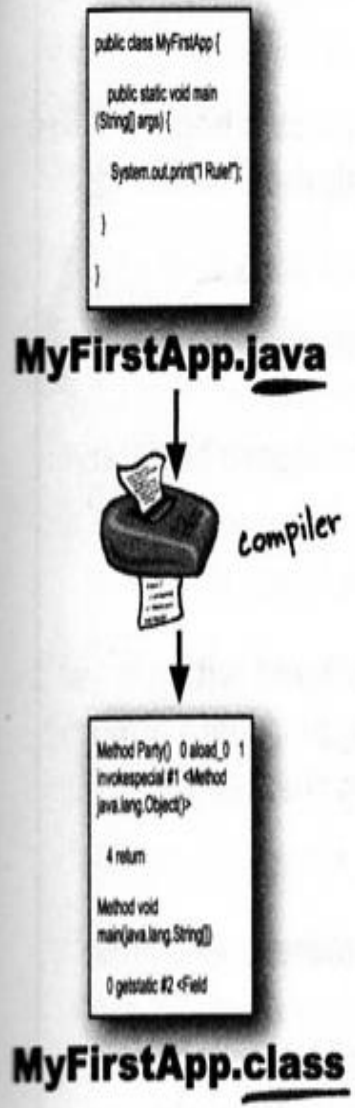
In Java, everything goes in a **class**. You'll type your source code file (with a *.java* extension), then compile it into a new class file (with a *.class* extension). When you run your program, you're really running a *class*.

Running a program means telling the Java Virtual Machine (JVM) to "Load the `Hello` class, then start executing its `main()` method. Keep running 'til all the code in main is finished."

In chapter 2, we go deeper into the whole *class* thing, but for now, all you need to think is, ***how do I write Java code so that it will run?*** And it all begins with **main**().

The **main**() method is where your program starts running.

No matter how big your program is (in other words, no matter how many *classes* your program uses), there's got to be a **main**() method to get the ball rolling.




```
public class MyFirstApp {

   public static void main (String[] args) {
     System.out.println("I Rule!");
     System.out.println("The World");
   }

}
```

❶ Save

MyFirstApp.java

❷ Compile

```
javac MyFirstApp.java
```

❸ Run

```
File Edit Window Help Scream
%java MyFirstApp
I Rule!
The World
```

JVM is an interpreter for bytecode

- Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments
- The reason is straightforward: only the JVM needs to be implemented for each platform
- When a program is interpreted, it generally runs substantially slower than it would run if compiled to executable code
- The use of bytecode enables the Java run-time system to execute programs much faster than you might expect

- Sun supplies its Just In Time (JIT) compiler for bytecode, which is included in the Java 2 release
- When the JIT compiler is part of the JVM, it compiles bytecode into executable code in real time, on a piece-by-piece, demand basis
- The JIT compiler compiles code as it is needed, during execution

The Java Buzzwords

- Simple
- Secure
- Portable
- Object-oriented
- Robust
- Multithreaded
- Architecture-neutral
- Interpreted
- High performance
- Distributed
- Dynamic

Simple

- If you already understand the basic concepts of object-oriented programming, learning Java will be even easier

- Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java

- Beyond its similarities with C/C++, Java has another attribute that makes it easy to learn: it makes an effort not to have *surprising* features

Object oriented

- The object model in Java is simple and easy to extend, while simple types, such as integers, are kept as high-performance nonobjects

Robust

- Ability to create robust programs was given a high priority in the design of Java

- To better understand how Java is robust, consider two of the main reasons for program failure: ***memory management mistakes*** and ***mishandled exceptional conditions*** (that is, run-time errors)

- Memory management can be a difficult, tedious task in traditional programming environments

- For example, in C/C++, the programmer must manually allocate and free all dynamic memory

- Programmers will either forget to free memory that has been previously allocated or, worse, try to free some memory that another part of their code is still using

- Java virtually eliminates these problems by managing memory allocation and deallocation for you

- Java provides object-oriented exception handling

Multithreaded

- Java supports multithreaded programming, which allows you to write programs that do many things simultaneously

- The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems

Architecture-Neutral

- Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction
- Java designers made several hard decisions in Java language and Java Virtual Machine in an attempt to alter this situation
- Their goal was "write once; run anywhere, any time, forever."

Interpreted and High Performance

- Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode
- This code can be interpreted on any system that provides a Java Virtual Machine
- the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler

Distributed

- Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols

- *Remote Method Invocation* (*RMI*) feature of Java brings an unparalleled level of abstraction to client/server programming

Dynamic

- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time.
- Makes it possible to dynamically link code in a safe and expedient manner.
- Small fragments of bytecode may be dynamically updated on a running system

AN OVERVIEW OF JAVA

- Object-oriented programming is at the core of Java
- all computer programs consist of two elements: ***code*** and ***data***
- A program can be conceptually organized around its code or around its data

- That is, some programs are written around "what is happening" and

  others are written around "who is being affected."

- The first way is called the ***process-oriented model***

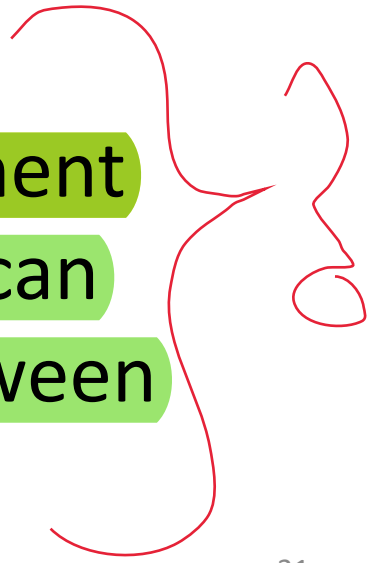
- The process-oriented model can be thought of as *code acting on data*

- The second approach, ***Object-oriented programming*** organizes a program around its data (that is, objects) and a set of well-defined interfaces to that data
- Characterized as *data controlling access to the code*

- Concept of ***Abstraction*** – focus is on essential features.

- Humans manage complexity through abstraction which is through the use of hierarchical classifications.

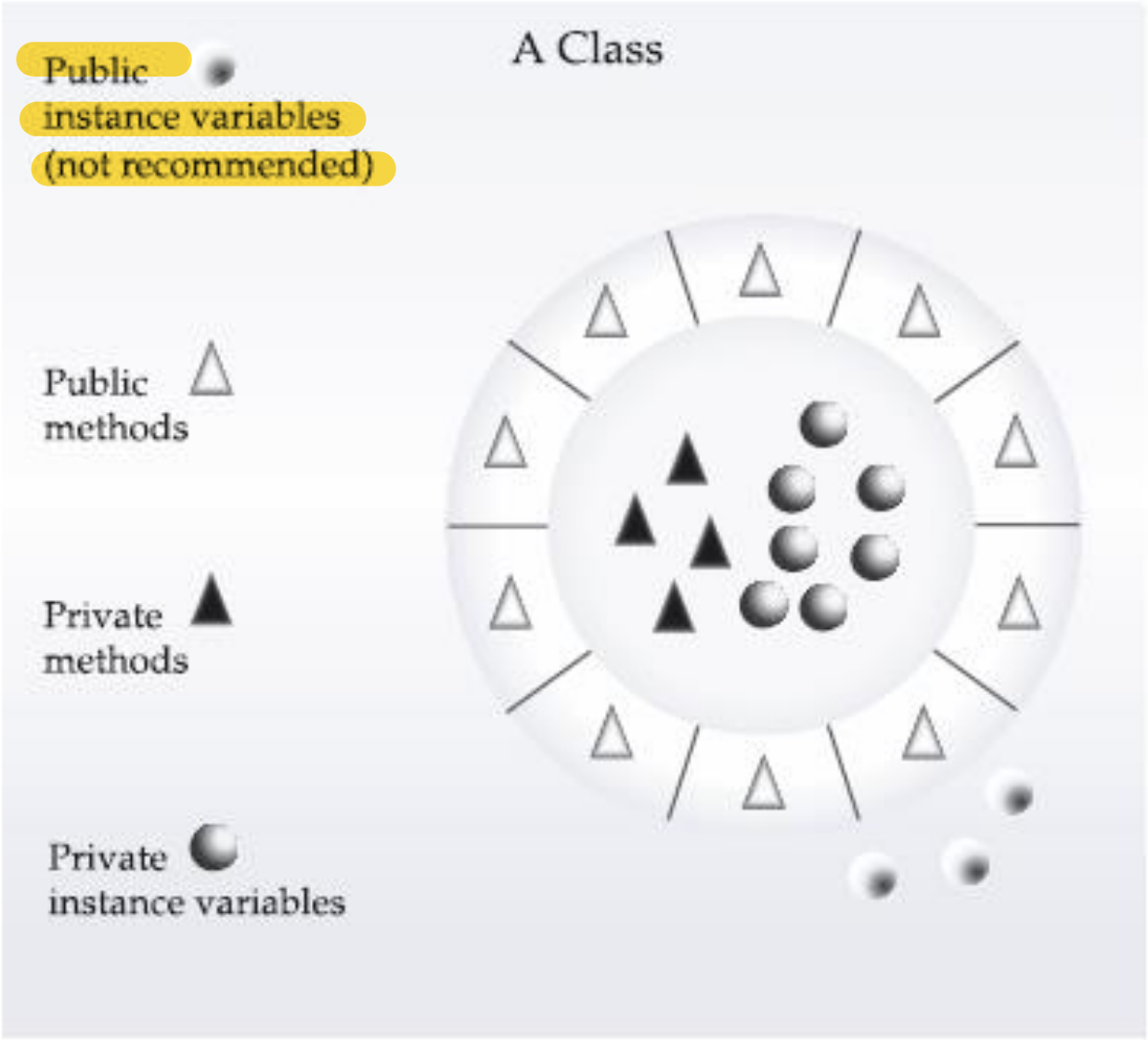
- Data can be transformed into component objects. A sequence of process steps can become a collection of messages between these objects - essence of OOP

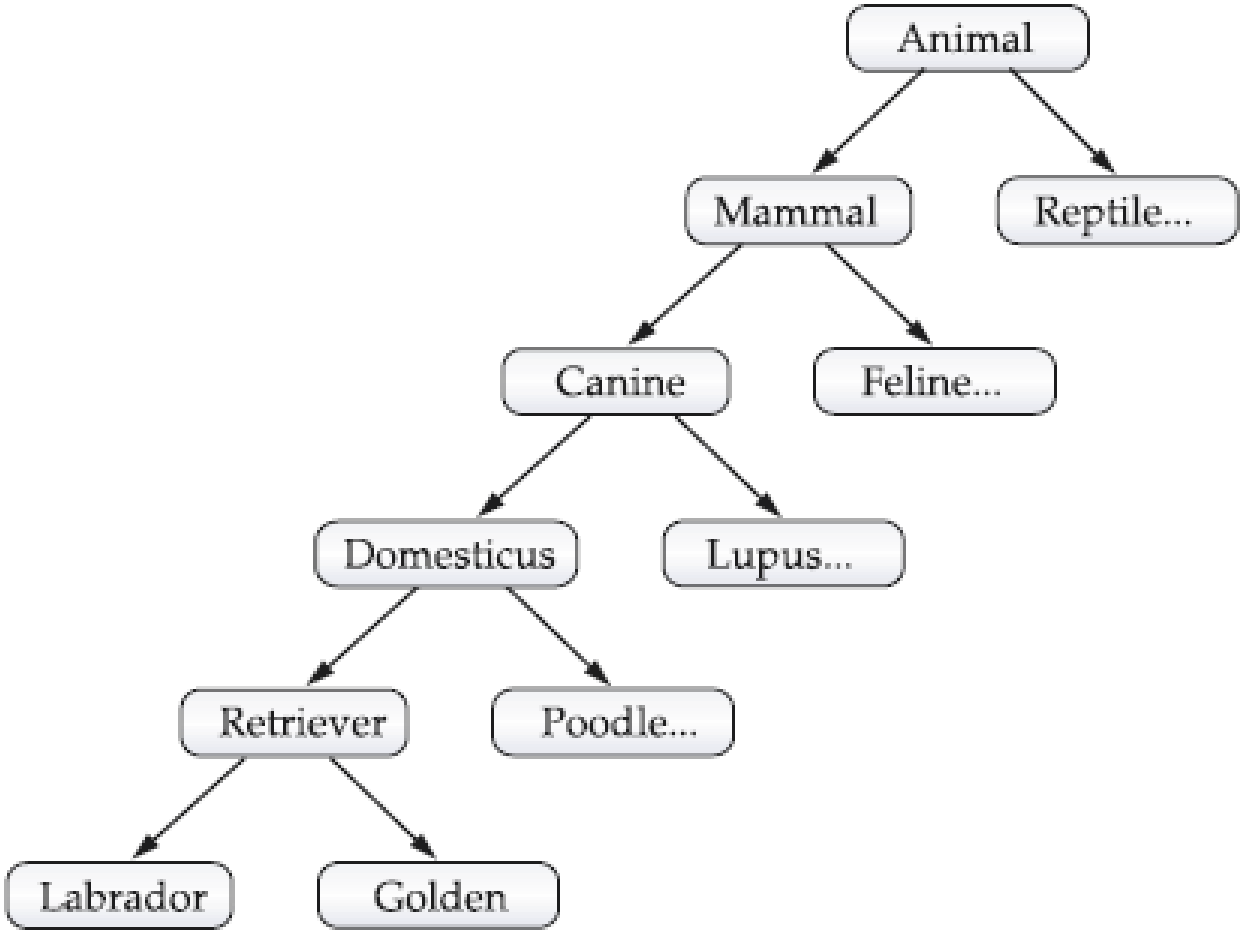


The Three OOP Principles:

- ***Encapsulation*** - is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse – class, member variables and methods

- ***Inheritance*** - the process by which one object acquires the properties of another object


A Class
Public
instance variables
(not recommended)
Public
methods
Private
methods
Private
instance variables


Animal
Mammal
Reptile...
Canine
Feline...
Domesticus
Lupus...
Retriever
Poodle...
Labrador
Golden

- ***Polymorphism*** - is a feature that allows one interface to be used for a general class of actions – " one interface , multiple methods"

- Polymorphism, encapsulation and inheritance works together - every java program involves these.

A First Simple Program

```
/* This is a simple Java program.
Call this file "Example.java".*/
class Example {
// Your program begins with a call to main().
public static void main(String args[]) {
System.out.println("This is a simple Java program.");
}
}
```

Compiling the Program

- ***C:\>javac Example.java***

- The javac compiler creates a file called ***Example.class*** that contains the bytecode version of the program

- The output of **javac** is not code that can be directly executed

- To actually run the program, you must use the Java interpreter, called **java**.

- ***C:\>java Example***

- When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared

- The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class

- The keyword **void** simply tells the compiler that **main( )** does not return a value

- **String[ ] args** declares a parameter named **args**, which is an array of instances of the class **String**. **args** receives any command-line arguments.

A Second Short Program

```
class Example2 {
public static void main(String args[]) {
int num; // this declares a variable called num
num = 100; // this assigns num the value 100
System.out.println("This is num: " + num);
num = num * 2;
System.out.print("The value of num * 2 is ");
System.out.println(num);
}
}
```

Two Control Statements

The if Statement:

if(*condition*) *statement*;

```
if(num < 100)
   System.out.println("num is less than 100");
```

The for Loop:

for(*initialization; condition; iteration*) *statement*;

```
class ForTest {
public static void main(String args[]) {
   int x;
for(x = 0; x<10; x = x+1)
   System.out.println("This is x: " + x);
}
}
```

Using Blocks of Code:

using { and }

Lexical Issues

- ***Whitespace:***
  - Java is a free-form language
  - In Java, whitespace is a space, tab, or newline

- ***Identifiers:***

  - Identifiers are used for class names, method names, and variable names

  - Java is case-sensitive

AvgTemp count a4 $test this_is_ok

Invalid variable names include:

2count high-temp Not/ok

Literals:

- A constant value in Java is created by using a *literal* representation of it

100	98.6	'X'	"This is a test"

Comments

- There are three types of comments defined by Java.
- Single-line and multiline
- The third type is called a *documentation comment*
- This type of comment is used to produce an HTML file that documents your program
- The documentation comment begins with a **/**** and ends with a ***/**

Separators

Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

The Java Keywords

- There are 50 reserved keywords currently defined in the Java language (enum)

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

DATA TYPES, VARIABLES, AND ARRAYS

Java Is a Strongly Typed Language

- Every variable has a type, every expression has a type, and every type is strictly defined
- All assignments, whether explicit or via parameter passing in method calls, are checked for type compatibility
- The Java compiler checks all expressions and parameters to ensure that the types are compatible

The Simple Types

Java defines eight simple (or elemental) types of data: **byte**, **short**, **int**, **long**, **char**, **float**, **double**, and **boolean**

Integers

Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

```
class Light {
public static void main(String args[]) {
    int lightspeed;
    long days;
    long seconds;          long distance;
    // approximate speed of light in miles per second
    lightspeed = 186000;
    days = 1000; // specify number of days here
    seconds = days * 24 * 60 * 60; // convert to seconds
    distance = lightspeed * seconds; // compute distance
    System.out.print("In " + days);
    System.out.print(" days light will travel about ");
    System.out.println(distance + " miles.");
}      }
```

Floating-Point Types

Floating-Point Data Type	Size (bits)*	Storage Requirement (bytes)	Default Value	Precision	Decimal Digits	Range	Accuracy
float	32	4	0.0f	Single	6 decimal digits	3.4e-038 to 3.4e+038	Low
double	64	8	0.0d	Double	15 decimal digits	1.7e-308 to 1.7e+308	High

```
class Area {
    public static void main(String args[ ]) {
        double pi, r, a;
        r = 10.8; // radius of circle
        pi = 3.1416; // pi, approximately
        a = pi * r * r; // compute area
        System.out.println("Area of circle is " + a);
    }
}
```

Characters

- **char** in Java is not the same as **char** in C or C++.
- In C/C++, **char** is an integer type that is 8 bits wide
- Instead, Java uses Unicode to represent characters
- *Unicode* defines a fully international character set that can represent all of the characters found in all human languages
  - E.g., '\u0061' – 'a'

- In Java **char** is a 16-bit type
- The range of a **char** is 0 to 65,536
- There are no negative **char**s

```
class CharDemo
{
    public static void main(String args[ ])
    {
        char ch1, ch2;
        ch1 = 88; // code for X
        ch2 = 'Y';
        System.out.print("ch1 and ch2: ");
        System.out.println(ch1 + " " + ch2);
    }
}
```

Output:

ch1 and ch2: X Y

```
class CharDemo2
{
    public static void main(String args[ ])
    {
        char ch1;
        ch1 = 'X';
        System.out.println("ch1 contains " + ch1);
        ch1++; // increment ch1
        System.out.println("ch1 is now " + ch1);
    }
}
```

Output:

ch1 contains X

ch1 is now Y

Booleans

- Can have only one of two possible values, **true** or **false**

```
class BoolTest {
    public static void main(String args[]) {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
    }
```

```
class BoolTest {
    public static void main(String args[]) {
        boolean b;
        b = false;
        System.out.println("b is " + b);
        b = true;
        System.out.println("b is " + b);
        // a boolean value can control the if statement
        if(b) System.out.println("This is executed.");
        b = false;
        if(b) System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value
        System.out.println("10 > 9 is " + (10 > 9));
    }
}
```

b is false
b is true
This is executed.
10 > 9 is true

- Integer Literals – Decimal, Hexa and Octal
  - In Java, binary starts with **0b**, octal starts with **0**, and hexadecimal starts with **0x**.
- Floating-Point Literals
  - For example, 2.0, 3.14159, and 0.6667 represent valid standard - notation floating-point numbers

  - *Scientific notation* uses a standard-notation, floating-point number plus a suffix that specifies a power of 10 by which the number is to be multiplied eg. 6.022E23, 314159E–05, and 2e+100

- Boolean Literals – true , false

- Character Literals
  - Characters in Java are indices into the Unicode character set
  - They are 16-bit values that can be converted into integers and manipulated with the integer operators, such as the addition and subtraction operators
  - A literal character is represented inside a pair of single quotes

Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\'	Single quote
\"	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

String Literals

- “Hello World”
- “two\n lines”
- “\”This is in quotes\””

Declaring a Variable

type identifier [= *value*] ;

```
int a, b, c;              // declares three ints, a, b, and c.
int d = 3, e, f = 5;      // declares three more ints, initializing
                          // d and f.
byte z = 22;              // initializes z.
double pi = 3.14159;      // declares an approximation of pi.
char x = 'x';             // the variable x has the value 'x'.
```

Dynamic Initialization

```
class DynInit {
public static void main(String args[]) {
double a = 3.0, b = 4.0;
// c is dynamically initialized
double c = Math.sqrt(a * a + b * b);
System.out.println("Hypotenuse is " + c);
}
}
```

The Scope and Lifetime of Variables

- Java allows variables to be declared within any block
- A block is begun with an opening curly brace and ended by a closing curly brace
- A block defines a *scope*

- In Java, the two major scopes are those defined by a class and those defined by a method

- As a general rule, variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope

- Scopes can be nested

- Objects declared in the outer scope will be visible to code within the inner scope ; but reverse not true

```
class Scope {
    public static void main (String args[]) {
        int x; // known to all code within main
        x = 10;
        if(x == 10) { // start new scope
            int y = 20; // known only to this block
            // x and y both known here.
            System.out.println("x and y: " + x + " " + y);
            x = y * 2;
        }
        // y = 100; // Error! y not known here
        // x is still known here.
        System.out.println("x is " + x);
    }
}
```

```
class LifeTime {
public static void main(String args[]) {
    int x;
    for(x = 0; x < 3; x++) {
        int y = -1; // y is initialized each time block is
                                entered
        System.out.println("y is: " + y);
        y = 100;
        System.out.println("y is now: " + y);
        }
    }
}
```

```
class ScopeErr {
    public static void main(String args[]) {
     int bar = 1;
     {
        int bar = 2;
     }
    }
}
```

```java
class Scope {
    public static void main(String args[]) {
        int x = 1;
        int y = 2;
        {
            int z = 3;
            y = 4;
            System.out.println("Inner block:");
            System.out.println(x);
            System.out.println(y);
            System.out.println(z);
        }
        System.out.println("Main block:");
        System.out.println(x);
        System.out.println(y);
        System.out.println(z);
        }
}
```

Type Conversion and Casting

- When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:
  - The two types are compatible.
  - The destination type is larger than the source type.

Byte –> Short –> Int –> Long – > Float –> Double

Widening or Automatic Conversion

```java
class AutoCast {

    // Main driver method
    public static void main(String[] args)
    {
        int i = 100;
        // Automatic type conversion
        // Integer to long type
        long l = i;
        // Automatic type conversion
        // long to float type
        float f = l;
        // Print and display commands
        System.out.println("Int value " + i);
        System.out.println("Long value " + l);
        System.out.println("Float value " + f);
    }
}
```

Output:

Int value 100

Long value 100

Float value 100.0

Casting Incompatible Types

- what if you want to assign an **int** value to a **byte** variable?

- This conversion will not be performed automatically, because a **byte** is smaller than an **int**

Datatype	Bits Acquired In Memory
boolean	1
byte	8 (1 byte)
char	16 (2 bytes)
short	16(2 bytes)
int	32 (4 bytes)
long	64 (8 bytes)
float	32 (4 bytes)
double	64 (8 bytes)

Narrowing or Explicit Conversion

(*target-type*) *value*

- *target-type* specifies the desired type to convert the specified value to

```
int a;
byte b;
// ...
b = (byte) a;
```

Double –> Float –> Long –> Int –> Short –> Byte

Narrowing or Explicit Conversion

```
class Conversion {
    public static void main(String args[])
    {
        // Declaring byte variable
        byte b;
        // Declaring and initializing integer and double
        int i = 128;
        double d = 323.142;
        // Display message
        System.out.println("Conversion of int to byte.");
        // i % 256
        b = (byte)i;
        // Print commands
        System.out.println("i = " + i + " b = " + b);
        System.out.println(
            "\nConversion of double to byte.");
        // d % 256
         b = (byte)d;
         System.out.println("d = " + d + " b= " + b);
     }
}
```

Output:

Conversion of int to byte.

i and b 257 1 (b is i%256)

Conversion of double to int.

d and i 323.142 323 (d is truncated to integer)

Conversion of double to byte.

d and b 323.142 67 (fractional component is lost and %256)

Automatic Type Promotion in Expressions

Example:

```
byte a = 40;
byte b = 50;
byte c = 100;
int d = a * b / c;
```

- Java automatically promotes each **byte** or **short** operand to **int** when evaluating an expression

- Subexpression **a * b** is performed using integer

```
byte b = 50
b = b * 2
// error: Cannot assign an int to a byte
```

In this case we need to explicitly specify:

```
byte b = 50;
b = (byte) (b*2);
```

The Type Promotion Rules

- All **byte** and **short** values are promoted to **int**
- If one operand is a **long**, the whole expression is promoted to **long**
- If one operand is a **float,** the entire expression is promoted to **float**
- If any of the operands is **double**, the result is **double**

```
class Promote {
public static void main(String args[ ]) {
    byte b = 42;
    char c = 'a';
    short s = 1024;
    int i = 50000;
    float f = 5.67f;
    double d = .1234;
    double result = (f * b) + (i / c) - (d * s);
    System.out.println((f * b) + " + " + (i / c) + " - " + (d * s));
    System.out.println("result = " + result);
    }
}
```

double result = (f * b) + (i / c) - (d * s);

- In the first sub expression, **f * b**, **b** is promoted to a **float** and the result of the sub expression is **float**

- Next, in the sub expression **i / c**, **c** is promoted to **int**, and the result is of type **int**

- In **d * s**, the value of **s** is promoted to **double**, and the type of the sub expression is **double**

- The outcome of **float** plus an **int** is a **float**

- Then the resultant **float** minus the last **double** is promoted to **double**, which is the type for the final result of the expression

Arrays

- An *array* is a group of like-typed variables that are referred to by a common name

- A specific element in an array is accessed by its index

One-Dimensional Arrays

- ***type var-name[ ];*** - No array exists

- ***array-var = new type[size];*** - allocating memory

Example:

int month_days[] = new int[12]

- Arrays can be initialized when they are declared

- An *array initializer* is a list of comma-separated expressions surrounded by curly braces

- There is no need to use **new**

```
class AutoArray {
public static void main(String args[ ]) {
    int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31,30
        , 31 };
    System.out.println("April has " + month_days[3] + " days.");
}
}
```

Output:
April has 30 days.

```
class Average {
public static void main(String args[]) {
    double nums[] = {10.1, 11.2, 12.3, 13.4, 14.5};
    double result = 0;
    int i;
    for(i=0; i<5; i++)
        result = result + nums[i];
    System.out.println("Average is " + result / 5);
}
}
```

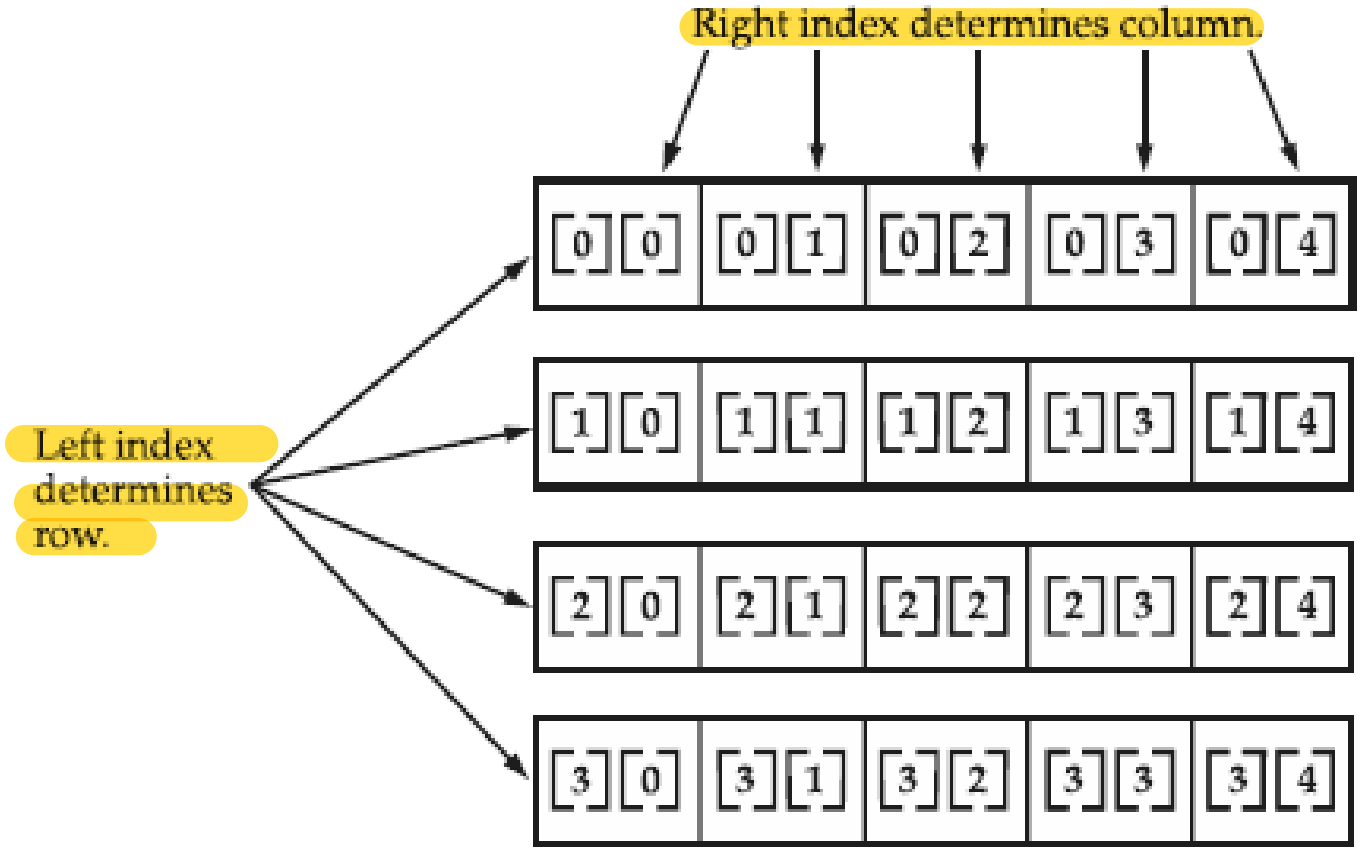
Multidimensional Arrays

- To declare a multidimensional array variable,

  specify each additional index using another set of square brackets

Example:

```
int twoD[][] = new int[4][5];
```



Given: int twoD [ ] [ ] = new int [4] [5] ;

```
class TwoDArray {
public static void main(String args[]) {
int twoD[][]= new int[4][5];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<5; j++) {
twoD[i][j] = k;
k++;
}
for(i=0; i<4; i++) {
for(j=0; j<5; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();
}           }         }
```

- When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension

- You can allocate the remaining dimensions separately

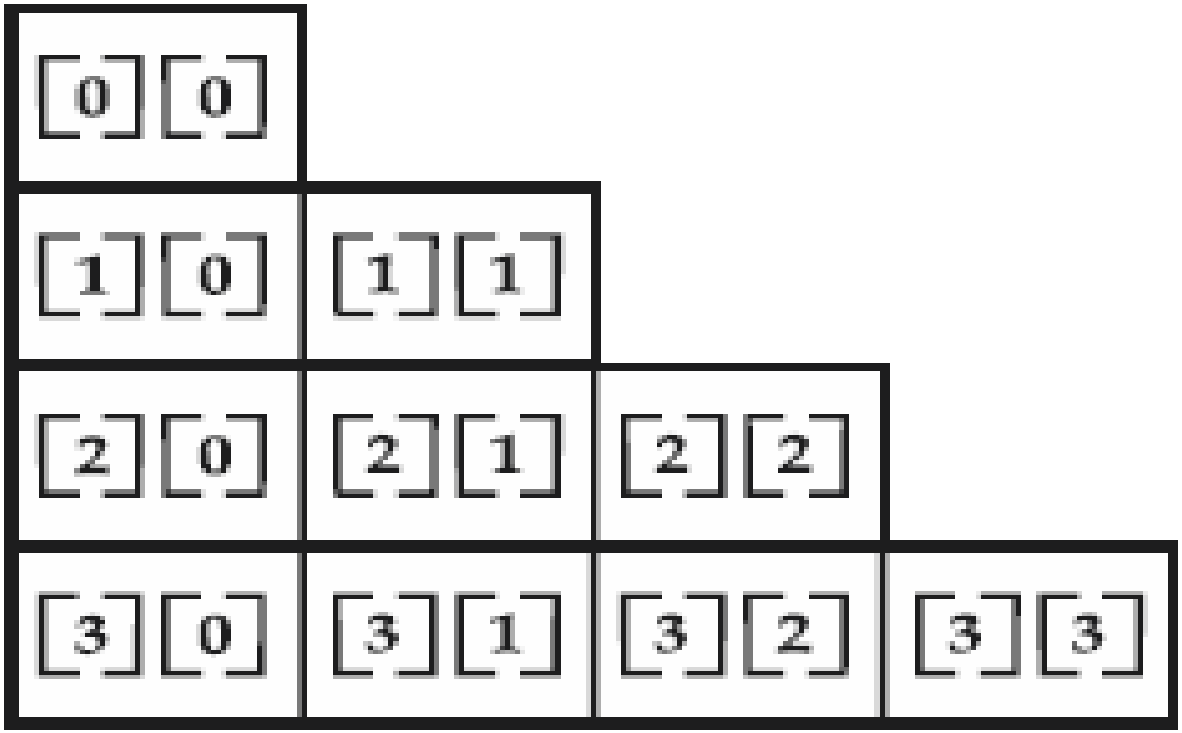
Example:

```
int twoD[][] = new int[4][];
twoD[0] = new int[5];
twoD[1] = new int[5];
twoD[2] = new int[5];
twoD[3] = new int[5];
```

- When you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension

- Since multidimensional arrays are actually arrays of arrays, the length of each array is under your control

```
class TwoDAgain {
public static void main(String args[]) {
int twoD[][] = new int[4][];
twoD[0] = new int[1];        twoD[1] = new int[2];
twoD[2] = new int[3];        twoD[3] = new int[4];
int i, j, k = 0;
for(i=0; i<4; i++)
for(j=0; j<i+1; j++) {
twoD[i][j] = k;        k++;        }
for(i=0; i<4; i++) {
for(j=0; j<i+1; j++)
System.out.print(twoD[i][j] + " ");
System.out.println();       }                  }
          }
```


[0] [0]
[1] [0] [1] [1]
[2] [0] [2] [1] [2] [2]
[3] [0] [3] [1] [3] [2] [3] [3]

- It is possible to initialize multidimensional arrays

- You can use expressions as well as literal values inside of array initializers

```
class Matrix {
public static void main(String args[]) {
double m[ ][ ] = {
{ 0*0, 1*0, 2*0, 3*0 },
{ 0*1, 1*1, 2*1, 3*1 },
{ 0*2, 1*2, 2*2, 3*2 },
{ 0*3, 1*3, 2*3, 3*3 }
};
int i, j;
for(i=0; i<4; i++) {
for(j=0; j<4; j++)
System.out.print(m[i][j] + " ");
System.out.println();       }                    }
          }
```

Output:

0.0 0.0 0.0 0.0

0.0 1.0 2.0 3.0

0.0 2.0 4.0 6.0

0.0 3.0 6.0 9.0

Alternative Array Declaration Syntax

- There is a second form that may be used to declare an array:

type*[ ] *var-name;

Example: ***These two are equivalent***

```
int al[ ] = new int[3];
int[ ] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[ ][ ] = new char[3][4];
char[ ][ ] twod2 = new char[3][4];
```

Note:

- Java does not support or allow pointers
- Java cannot allow pointers, because doing so would allow Java applets to breach the firewall between the Java execution environment and the host computer
- Java is designed in such a way that as long as you stay within the confines of the execution environment, you will never need to use a pointer, nor would there be any benefit in using one

OPERATORS

Arithmetic Operators

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Arithmetic Assignment Operators

- a = a + 4;

- a += 4;

Any statement of the form

var = var op expression;

can be rewritten as

var op= expression;

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	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;

```
1  class BasicMath {
2  public static void main(String args[]) {
3   // arithmetic using integers
4       System.out.println("Integer Arithmetic");
5       int a = 1 + 1;              int b = a * 3;
6       int c = b / 4;              int d = c - a;
7       int e = -d;
8       System.out.println("a = " + a);
9       System.out.println("b = " + b);
10      System.out.println("c = " + c);
11      System.out.println("d = " + d);
12      System.out.println("e = " + e);
```

```
13  // arithmetic using doubles
14      System.out.println("\nFloating Point Arithmetic");
15      double da = 1 + 1;       double db = da * 3;
16      double dc = db / 4;      double dd = dc - a;
17      double de = -dd;
18      System.out.println("da = " + da);
19      System.out.println("db = " + db);
20      System.out.println("dc = " + dc);
21      System.out.println("dd = " + dd);
22      System.out.println("de = " + de);
23      }
24  }
```

- The modulus operator, **%**, returns the remainder of a division operation

- It can be applied to floating-point types as well as integer types

- This differs from C/C++, in which the **%** can only be applied to integer types

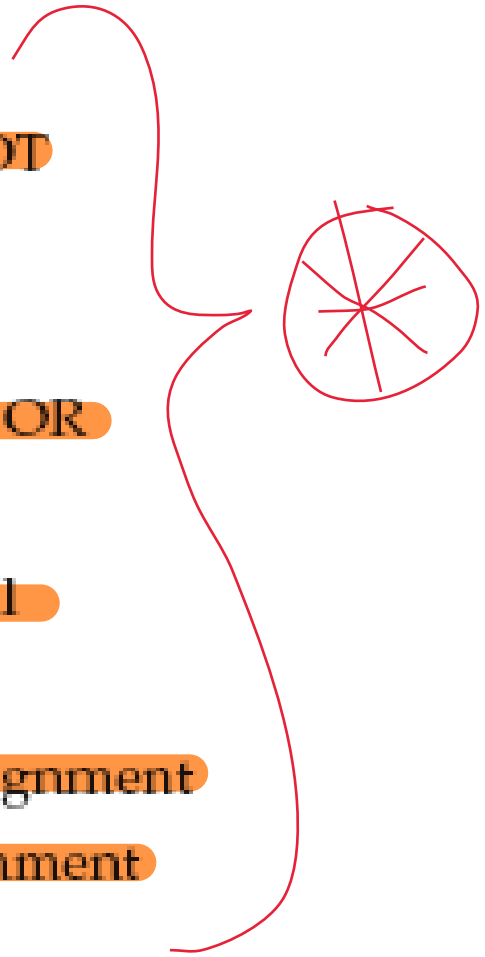
```
class Modulus {
public static void main(String args[]) {
int x = 42;
double y = 42.25;
System.out.println("x mod 10 = " + x % 10);
System.out.println("y mod 10 = " + y % 10);
}  }
```

When you run this program you will get the following output:

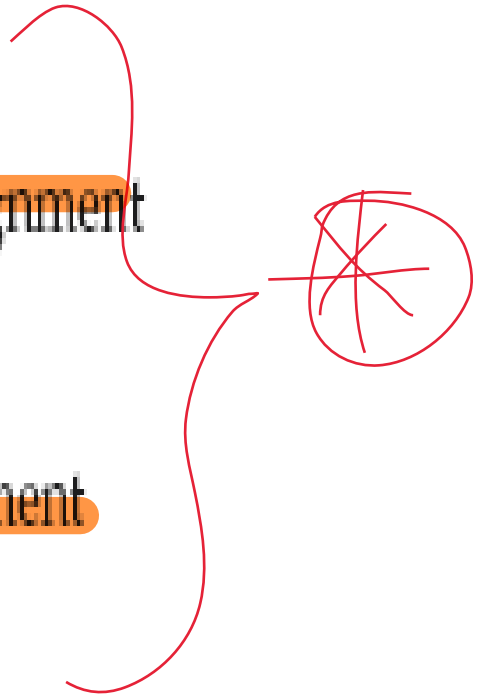
```
x mod 10 = 2
y mod 10 = 2.25
```

The Bitwise Operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment



Operator	Result
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment



A	B	A | B	A & B	A ^ B	~A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
class BitLogic {
public static void main(String args[]) {
String binary[ ] = {  "0000", "0001", "0010", "0011",
    "0100", "0101", "0110", "0111", "1000", "1001",
    "1010", "1011", "1100", "1101", "1110", "1111" };
int a = 3; // 0 + 2 + 1 or 0011 in binary
int b = 6; // 4 + 2 + 0 or 0110 in binary
int c = a | b;              int d = a & b;
int e = a ^ b;              int f = (~a & b) | (a & ~b);
int g = ~a & 0x0f;
System.out.println(" a = " + binary[a]);
System.out.println(" b = " + binary[b]);
System.out.println(" a|b = " + binary[c]);
System.out.println(" a&b = " + binary[d]);
```

to get 12 varna 1100 means -4 in 2's Complement

```
System.out.println(" a^b = " + binary[e]);
System.out.println("~a&b|a&~b = " + binary[f]);
System.out.println(" ~a = " + binary[g]);   }    }
```

- Output of the program :

  a = 0011

  b = 0110

  a | b = 0111

  a & b= 0010

  a ^ b = 0101

  ~a&B | a&~b = 0101

  ~a = 1100

The Left Shift

It has this general form:

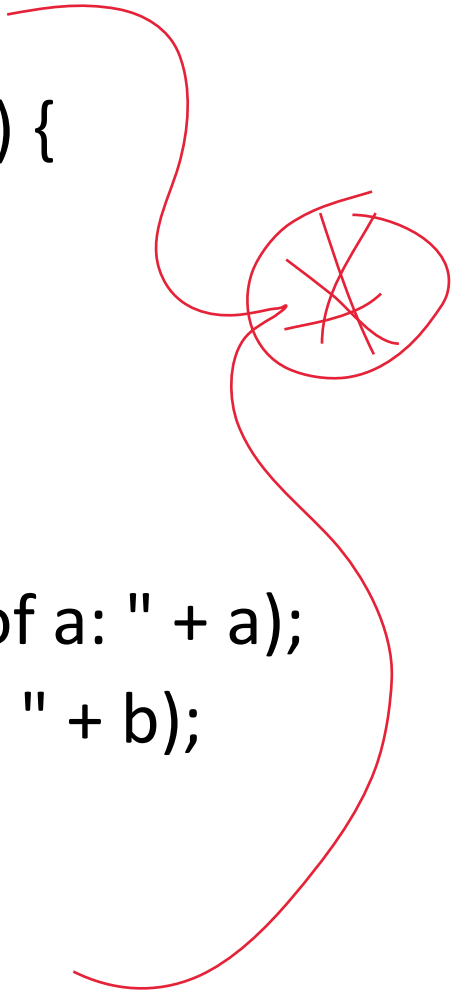
value << num

- If you left-shift a **byte** value, that value will first be promoted to **int** and then shifted

- This means that you must discard the top three bytes of the result if what you want is the result of a shifted **byte** value

- The easiest way to do this is to simply cast the result back into a **byte**

```
class ByteShift {
public static void main(String args[]) {
byte a = 64, b;
int i;
i = a << 2;
b = (byte) (a << 2);
System.out.println("Original value of a: " + a);
System.out.println("i and b: " + i + " " + b);
}   }
```



Output:

Original value of a*: *64

i and b: 256 0

The Right Shift

- The right shift operator, **>>**, shifts all of the bits in a value to the right a specified number of times

value >> num

```
int a = 32;
a = a >> 2; // a now contains 8
```

```
int a = 35;
a = a >> 2; // a still contains 8
```

11111000 –8

>>1

11111100 –4

- Sign extension :- when you are shifting right, the left most bits exposed by the right shift are filled in with the contents of the top bit. If you shift -1, the result always remains -1.
- In the program, the right shifted value is masked by ANDing it with 0x0f to discard the sign extended bits so that the value can be used an index in to the array of hexadecimal characters.

Bitwise Operator Assignments

```
a = a >> 4;
a >>= 4;

a = a | b;
a |= b;
```

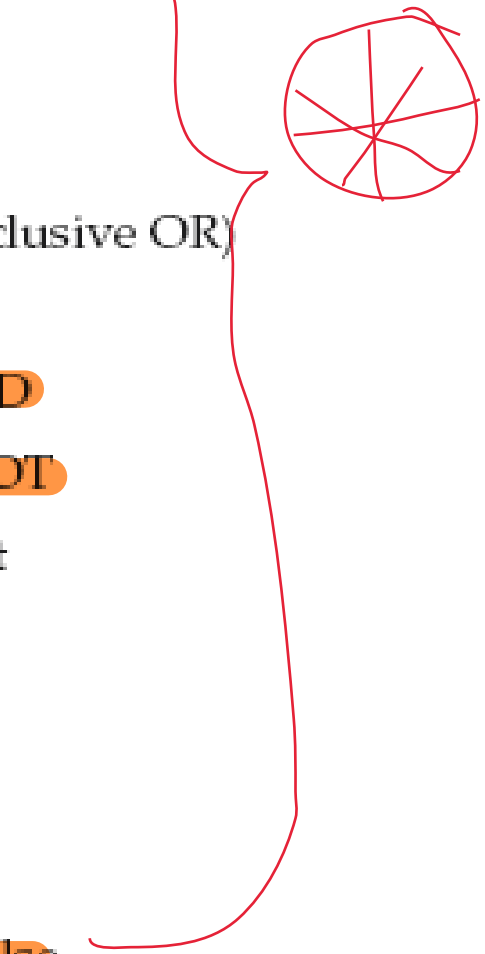
Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

In Java, ***true*** *and* ***false*** *are nonnumeric values which do not relate to zero or nonzero*

Boolean Logical Operators

Operator	Result
&	Logical AND
|	Logical OR
^	Logical XOR (exclusive OR)
||	Short-circuit OR
&&	Short-circuit AND
!	Logical unary NOT
&=	AND assignment
|=	OR assignment
^=	XOR assignment
==	Equal to
!=	Not equal to
?:	Ternary if-then-else



A	B	A | B	A & B	A ^ B	!A
False	False	False	False	False	True
True	False	True	False	True	False
False	True	True	False	True	True
True	True	True	True	False	False

Short-Circuit Logical Operators

- Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone

Example 1:

```
if (denom != 0 && num / denom > 10)
```

Example 2:

```
if(c==1 & e++ < 100) d = 100;
```

The Assignment Operator

var = expression;

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

The ? Operator

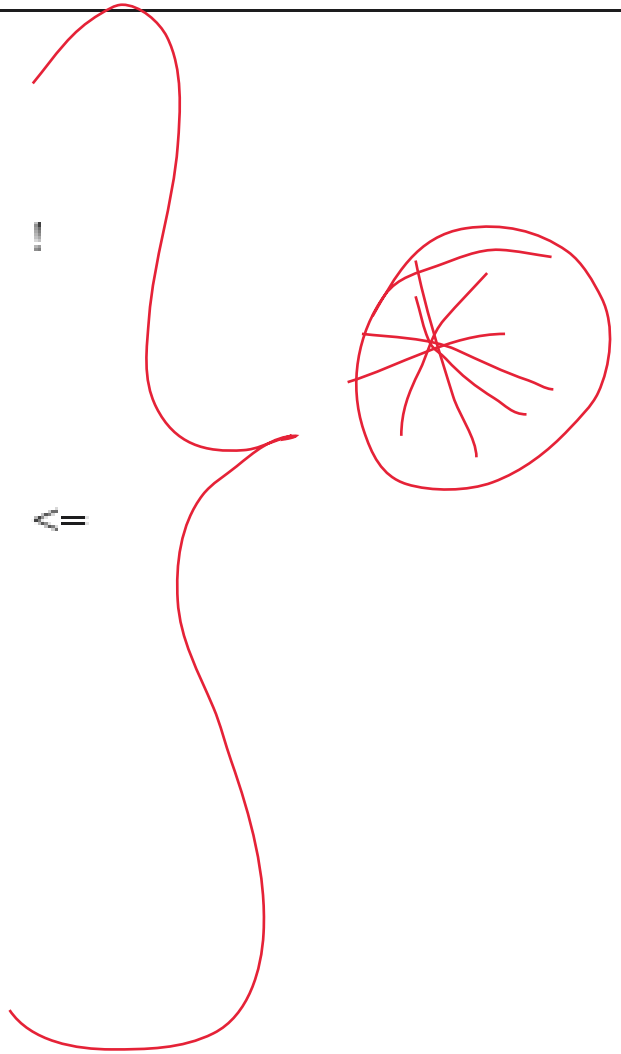
expression1 **?** *expression2* **:** *expression3*

Example:

```
ratio = denom == 0 ? 0 : num / denom;
```

Operator Precedence

Highest			
( )	[ ]	.	
++	−−	~	!
*	/	%	
+	−		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
||			
?:			
=	op=		
Lowest			



```
1  class Precedence {
2      public static void main(String[] args) {
3
4          int a = 10, b = 5, c = 1, result;
5          result = a-++c-++b;
6
7          System.out.println(result);
8      }
9  }
```

- ***Parentheses (redundant or not) do not degrade the performance of your program***

- ***Therefore, adding parentheses to reduce ambiguity does not negatively affect your program***

Chapter 4

Introducing Classes, Objects, and Methods

Class Fundamentals

- Classes are templates or blueprints that specify how to build objects.
- Objects are instances of a class.
- A class can be used to create any number of objects, all of the same form, but possibly containing different data.
- Well-designed classes group logically connected data with methods for acting on that data.

Class General Form

```
class classname {
    // declare instance variables
    type varname;

    // declare constructors
    classname( parameters ) {
      // body of constructor
    }

    // declare methods
    type methodname( parameters ) {
      // body of method
    }
}
```

Class General Form explanation

- Collectively, the methods and variables defined within a class are called
  - *members* of the class
- Variables defined within a class are called instance variables
  - because each instance of the class contains its own copy of these variables

Example of a Class

- **Vehicle** class declaration:

```
class Vehicle {
int passengers, fuelCap, mpg;
}
```

- **VehicleDemo** class declaration:

```
class VehicleDemo{
public static void main(String[] args) {
 Vehicle van = new Vehicle();
 Vehicle car = new Vehicle();
  car.mpg = 25;
  car.fuelCap = 12;
}
 }
```

Object Creation

- First, you must declare a variable of the class type
- This variable does not define an object ; Instead, it is simply a variable that can *refer* to an object
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the **new** operator
- The new operator dynamically allocates memory for an object and returns a reference to it:

```
Vehicle van ; // declare reference to object
van = new Vehicle(); // allocate a Vehicle object
```

OR

```
Vehicle van = new Vehicle();
```

Reference Variables

- The local variables **van** and **car** refer to different objects.
- Both objects have the same form (with 3 instance variables **passengers**, **fuelCap**, and **mpg**), but they have their own copies of the 3 instance variables.
- To access instance variables and methods, use the dot (.) operator:

```
car.mpg = 25;
car.fuelCap = 12;
```

Assignment of References

- If you assign

  ```
  van = car;
  ```

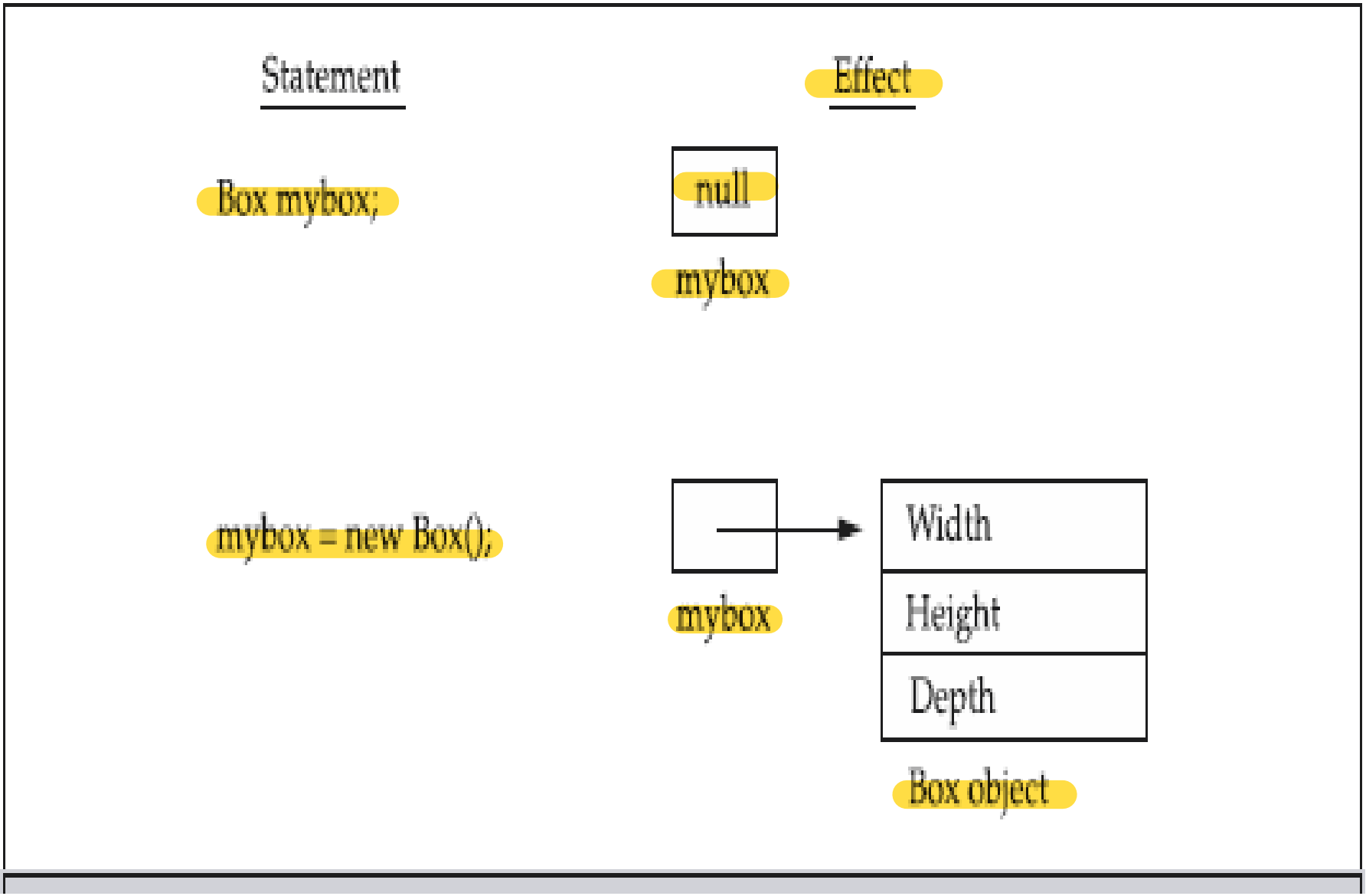
  then both variables refer to the same object.
- In that case, if you change an instance variable's value in **van**, it will change the value in **car** as well, since they refer to the same object.
- The object that **van** previously referred to is garbage collected if no other references to the object exist.

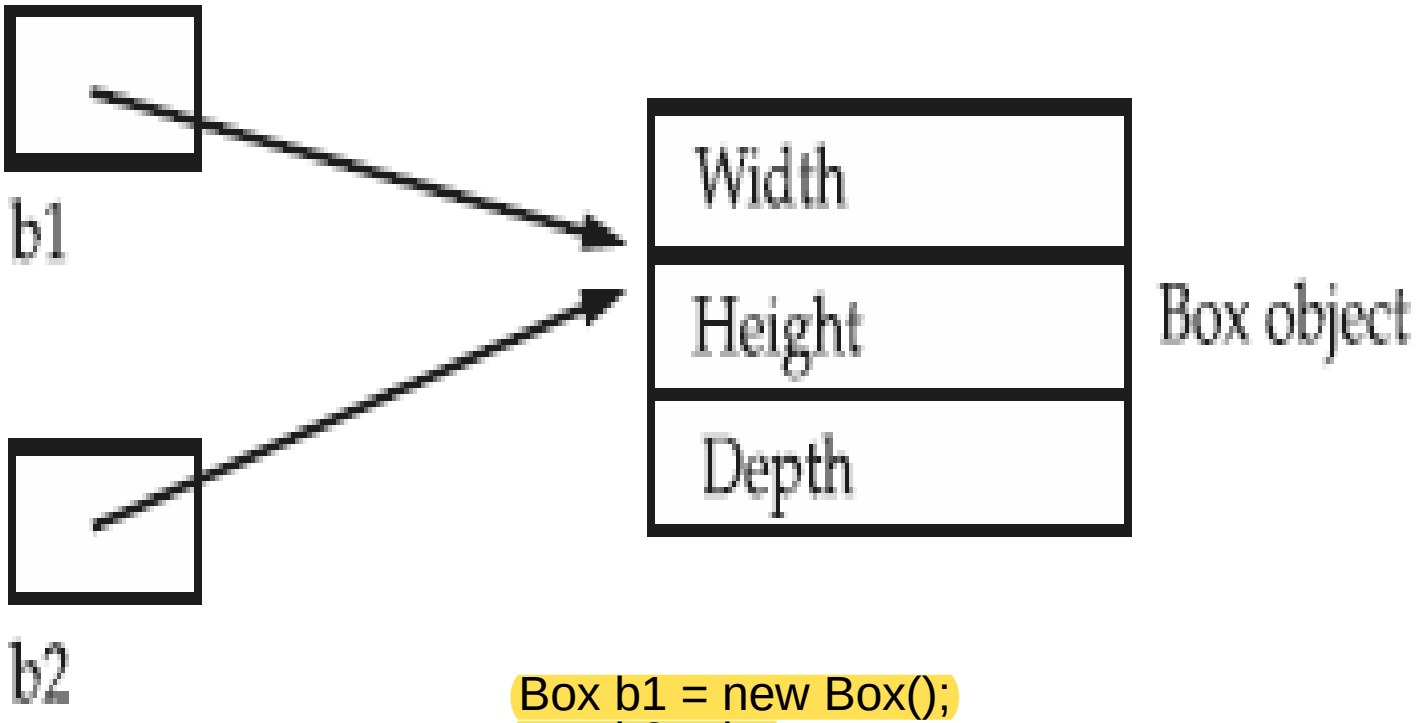
Assignment of References

Note:

When you assign one object reference variable to another object reference variable,

you are not creating a copy of the object, you are only making a copy of the reference


Statement
Effect
Box mybox;
null
mybox
mybox = new Box();
Width
Height
Depth
mybox
Box object



Box b1 = new Box();
Box b2 = b1

Methods

- General form:

  return-type methodname (parameters) {

  statements;

  }

- Parameters are local variables that receive their values from the caller of the method.
- The return type can be any valid type or **void** if the method doesn't return a value.

Example Method

- The following method can be added to the **Vehicle** class:

```
void range() {
  System.out.println("range: " + fuelCap * mpg);
}
```

- If the **VehicleDemo** class's **main( )** method calls

```
car.range();
```

then "range: 300" will be displayed.

Returning from a **void** Method

- Two ways to return from a **void** method:
  - when the method's closing brace is encountered
  - when a **return** statement is encountered
- Examples:

```
void sayHello() {
  System.out.println("Hello");
}

void sayHello() {
  System.out.println("Hello");
  return;
}
```

Returning a Value

- To return a value from a method, you must use a **return** statement of the form:

  return *value*;

- Example:
  - In the **Vehicle** class:

```
int range() {
  return fuelCap * mpg;
}
```

  - In **VehicleDemo** class's **main( )** method:

```
int range = car.range();
System.out.println(range); // prints "300"
```

Using Parameters

- A *parameter* is a variable whose scope is the method body and whose initial value is specified by the caller.
- Example:
  - In the **Vehicle** class:

```
double fuelNeeded(int distance) {
   return (double) distance / mpg;
}
```

  - In **VehicleDemo**'s **main( )** method:

```
System.out.println(car.fuelNeeded(750));
// prints "30.0"
```

Constructors

- Used to initialize an object when it is created
- Syntactically it is like a method.
- Its name is the name of the class.
- Have no return type, not even void
- Automatically called when an object is created.
- Example constructor for **Vehicle** class:

```
Vehicle() {
  fuelCap = 12;
  mpg = 25;
  passengers = 5;
}
```

Default Constructors

Consider

Vehicle van = new Vehicle();

- **new Vehicle()** is calling the **Vehicle()** constructor
- When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class
- The **default constructor** automatically **initializes all instance variables to zero**

Constructors with Parameters

- Example:
  - In the **Vehicle** class:

```
Vehicle(int p, int f, int m) {
  passengers = p; fuelCap = f; mpg = m;
}
```

  - In the **VehicleDemo**'s **main( )** method:

```
Vehicle car = new Vehicle(5, 12, 25);
Vehicle van = new Vehicle(7, 24, 21);
```

The Keyword *this*

- **this** is an implicit argument that refers to the object on which the method is called.
- **this** is useful for making it clear that you are referring to an instance variable.
- Example in **Vehicle** class:

```
double fuelNeeded(int distance) {
   return (double) distance / this.mpg;
}
```

Instance variable hiding

- It is illegal in java to declare two local variables with the same name inside the same or enclosing scopes.
- Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class instance variables. We can see the following example for this explanation.

Example:

```
Vehicle(int p, int f, int m) {
    passengers = p; fuelCap = f; mpg = m;
  }
```

Instance variable hiding

- However, when a local variable has the same name as an instance variable, the **local variable hides the instance variable.** This is why passengers, fuelCap, mpg were not used as the names for the parameters to the Vehicle() constructor inside the Vehicle class. If they had been, then **passengers** would have referred to the formal parameter, hiding the **instance variable passengers.**

Example:

```
Vehicle(int passengers, int f, int m) {
   passengers = passengers; // passengers refers to formal parameter
fuelCap = f;
 mpg = m;
 }
```

Instance variable hiding

- While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any name space collisions that might occur between instance variables and local variables.

```
// Use this to resolve name-space collisions.
Vehicle(double passengers, double fuelCap, double mpg) {
        this. passengers = passengers;
        this. fuelCap= fuelCap;
        this.mpg = mpg;
    }
```

Progress Check

1. What is the difference between a class and an object?
2. How is class defined?
3. What two things does a class contain?
4. What does new do?
5. Each object has its own copies of the class's _____
6. What is *this*?
7. When is a constructor executed?
8. Does a constructor have a return type?
9. Can a constructor have one or more parameters?
10. What happens when one reference variable is assigned to another?

Garbage Collection

- In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator

- Java takes a different approach; it handles deallocation for you **automatically**
- The technique that accomplishes this is called **garbage collection**

Garbage Collection

- When no references to an object exist,
  - that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed

- There is **no explicit code** needed to destroy objects as in C++

The finalize( ) Method

- Sometimes an object will need to perform some action when it is destroyed

- For example, if an object is holding some non-Java resource such as a file handle or window character font,
  - then you might want to make sure these resources are freed before an object is destroyed

The finalize( ) Method

- By using finalization, you can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector
- To add a finalizer to a class, you simply define the **finalize( )** method
- Inside the **finalize( )** method you will specify those actions that must be performed before an object is destroyed

The finalize( ) Method

- The finalize( ) method has this general form:

```
protected void finalize( )
{
// finalization code here
}
```

- The keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class

The finalize( ) Method

- It is important to understand that finalize( ) is only called just prior to garbage collection
- It is not called when an object goes out-of-scope, for example
- This means that you cannot know when—or even if—finalize( ) will be executed
- Therefore, your program should provide other means of releasing system resources, etc., used by the object

Exercise

Q1. Define a class to represent a complex number called **Complex.** Provide the following member functions:-

i. To assign initial values to the Complex object (using constructor).

ii. To display a complex number in a+ib format.

Write a main function to test the class.

Q2. Create a Swapper class with two integer instance variables x and y and a constructor with two parameters that initialize the two variables. Also include three methods: a getX( ) method that returns x, a getY( ) method that returns y, and a void swap( ) method that swaps the values of x and y. Then create a SwapperDemo class that tests all the methods

Chapter 5

More Data Types and Operators

Arrays

- An *array* is a collection of variables of the same type referred to by a common name.
- Each of the variables is specified by an index.
- One way to declare an array:
  type[ ] *arrayname* = new *type*[*size*];
- Example:

```
int[] grades = new int[30];
```

- In the example, **grades** is the name of the collection of 30 integer variables.

Accessing Array Variables

- You must specify an index to access a variable.
- If the array has size 30, the indices are 0 to 29.
- Example:

```
int[] grades = new int[30];
grades[0] = 100;
grades[29] = 0;
grades[1] = grades[0];
grades[2] = grades[3*2+1];
for(int i = 0; i < 30; i++)
  grades[i] = i+70;
```

Array Initializers

- To create and initialize an array at the same time, you can use this form:
  type[ ] *arrayname* = { *val1*, *val2*, …, *valN* };
- Example:

```
int[] fourVals = {3, 1, 4, 1};
```

- This example creates an array of length 4 storing the four values 3, 1, 4, and 1.

Bubble Sort

- You can sort an array **nums** of length **size** as follows:

```
for(int a = 0; a < size; a++)
   for(int b = 1; b < (size-1); b++) {
      if(nums[b-1] > nums[b]) {
         // if out of order exchange elements
        int t = nums[b-1];
        nums[b-1] = nums[b];
        nums[b] = t;
   }
}
```

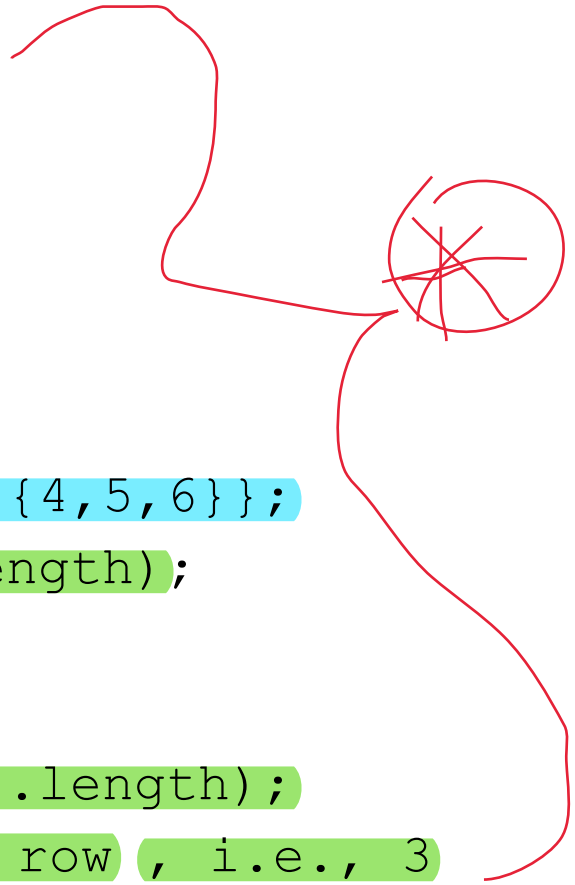
MultiDimensional Arrays:

- Two-Dimensional Arrays :
  An array of one-dimensional arrays.
- Examples:

```
int[][] table = new int[3][4];
table[0][1] = 3;
for(int i = 0; i < 3; i++)
  for(int j = 0; j < 4; j++)
    table[i][j] = i+j;

int[][] newTable = {{1,2},{3,4},{5,6}};
```

Irregular/ragged Two-dimensional Arrays

```
int[][] data = new int[3][];
data[0] = new int[1];
data[1] = new int[2];
data[2] = new int[4];

int[][] moreData = {{1},{2,3},{4,5,6}};
System.out.println(moreData.length);
// displays no. of rows 3

System.out.println(moreData[2].length);
// displays no. of cols in 2nd row , i.e., 3
```



Arrays of Three or More Dimensions

The general form of a multidimensional array declaration:

- type [ ][ ]...[ ] name = new type[size1][size2]...[sizeN];
- **For example**, the following declaration creates a 4 × 10 × 3 three-dimensional integer array.
- Int [][][] multidim = new int[4][10][3];

Initializing Multidimensional Arrays

- Enclosing each dimension's initializer list
  - within its own set of curly braces.

For example, the general form of array initialization for a two-dimensional array is shown here:

```
type-specifier array_name[ ] [ ] = {
{ val, val, val, ..., val },
{ val, val, val, ..., val },
}
```

Here, val indicates an initialization value.

Alternative Array Declaration Syntax

- *type*[ ] *var-name*;

The square brackets follow the type specifier, not the name of the array variable

- int counter[] = new int[3];
- int[] counter = new int[3];

The following declarations are also equivalent:

- char table[][] = new char[3][4];
- char[][] table = new char[3][4];

Alternative Array Declaration Syntax

- Convenient when declaring several arrays at the same time. For example, to create three arrays
- **int[] nums, nums2, nums3;**
- This creates three array variables of type **int**. It is the same as writing
- **int nums[], nums2[], nums3[];**

also, create three arrays

Alternative Array Declaration Syntax

- useful when specifying an array as a return type for a method. For example,
- **int[] someMeth( ) { ... }**
- This declares that **someMeth( )** returns an array of type **int**.

Assigning Array References

- int nums1[] = new int[10];
- int nums2[] = new int[10];
- for(i=0; i < 10; i++) nums1[i] = i;
- for(i=0; i < 10; i++) nums2[i] = -i;

- Assign an array reference
- **nums2 = nums1;** // now nums2 refers to nums1

Assigning Array References

- nums1: 0 1 2 3 4 5 6 7 8 9
- nums2: 0 -1 -2 -3 -4 -5 -6 -7 -8 -9

- nums2 after assignment: 0 1 2 3 4 5 6 7 8 9
- nums1 after change through nums2: 0 1 2 3 4 5 6 7 8 9

Using the length Member

- All arrays have a read-only instance variable called **length** that contains the number of elements that the array can hold.

Using the length Member: An Example

```
int list[] = new int[10]; int nums[] = { 1, 2, 3 };
// a variable-length table
int table[][] = { {1, 2, 3},{4, 5}, {6, 7, 8, 9} };
SOP("length of list is " + list.length);
SOP("length of nums is " + nums.length);
SOP("length of table is " + table.length);
SOP("length of table[0] is " + table[0].length);
SOP("length of table[1] is " + table[1].length);
SOP("length of table[2] is " + table[2].length);
```

Using the length Member: An Example

```
int list[] = new int[10]; int nums[] = { 1, 2, 3 };
// a variable-length table
int table[][] = { {1, 2, 3},{4, 5}, {6, 7, 8, 9} };
SOP("length of list is " + list.length); //10
SOP("length of nums is " + nums.length);//3
SOP("length of table is " + table.length);//3
SOP("length of table[0] is " + table[0].length);//3
SOP("length of table[1] is " + table[1].length);//2
SOP("length of table[2] is " + table[2].length);//4
```

Exercise

- Implement stack data structure using Java program.
  - Create a class name called *Stack*
  - Declare an integer array for the stack.
  - An integer for storing the top of stack(top)
  - Push method should insert an element into the stack.
  - Pop method should return the stack *top* element.
  - Both Push and Pop should be checked for Stack overflow and underflow error.

The for-each Style Loop

- General form:

  for(*type iterVar* : *collection*) *statement-block*

- *type* specifies the type, and *iterVar* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end.
- Its iteration variable is "read-only" as it relates to the underlying array.
- The contents of the array can't be changed by assigning the iteration variable a new value.

The **for-each** Style Loop

- Example:

```
   int[] data = {3,4,5,6};
// these two loops do the same thing
 for(int i = 0; i < data.length; i++)
     System.out.println(data[i]);

  for(int i : data)
    System.out.println(i);
  // no subscript needed
```

Use for-each style for to display and sum the values of an Array

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
 int sum = 0; //
 for(int x : nums)
{  System.out.println("Value is: " + x);
    sum += x;
}
 System.out.println("Summation: " + sum);
```

The **for-each** Style Loop

```
//To add only the first 5 elements.
int nums[]={ 1,2,3,4,5,6,7,8,9};
for(int x : nums)
{   SOP("Value is: " + x);
    sum += x; // stop the loop when 5 is obtained
    if(x == 5)
        break;
} System.out.println("Summation: " + sum);
```

The **for-each** Style Loop

```
int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
for(int x : nums)
 {  System.out.print(x + " ");
    x = x * 10;
 // no effect on nums and doesnot change nums
}
```

Use for-each style for on a two-dimensional array

```
int sum = 0;  int nums[][] = new int[3][5];
for(int i = 0; i < 3; i++) // give nums some values
 for(int j=0; j < 5; j++)
     nums[i][j] = (i+1)*(j+1);
// Use for-each for loop to display and sum the values
for(int x[] : nums) //Notice how x is declared
{ for(int y : x)
  { System.out.println("Value is: " + y);
    sum += y;
  }
 } System.out.println("Summation: " + sum);
```

Applying the Enhanced for loop

- program using **for each** loop to **search an unsorted array** for a value. It stops if the value is found.

```
int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4 }; int val = 5;
boolean found = false;
for(int x : nums)
 {   if(x == val)
        { found = true; break;   }
}
if(found)
   System.out.println("Value found!");
```

Constructing Strings

- Java strings are objects of class **String**.
- They are constructed various ways.
- Example:

```
// all 3 statements create new String objects
String s1 = "hello";
String s2 = new String("hai");
String s3 = new String(s2);
```

if no parameter given then gives null string

boolean equals(str)	Returns true if the invoking string contains the same character sequence as str.
int length( )	Returns the number of characters in the string.
char charAt(index)	Returns the character at the index specified by index.
int compareTo(str) s - str	Returns a negative value if the invoking string is less than str, a positive value if the invoking string is greater than str, and zero if the strings are equal.
int indexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the first match or -1 on failure.
int lastIndexOf(str)	Searches the invoking string for the substring specified by str. Returns the index of the last match or -1 on failure.

Examples Using String Operations

```
String s1 = "abcde";
System.out.println(s1.length()); // prints "5"
System.out.println(s1.charAt(2)); // prints "c"
if(s1.compareTo("xyz") < 0)
  System.out.println("Yes"); // prints "Yes"
System.out.println(s1.indexOf("bc")); // prints "1"
System.out.println(s1.indexOf("f")); // prints "-1"
System.out.println(s1.indexOf("ef")); // prints "-1"
```

String Properties

- **Strings** are immutable;
- characters of a String can be accessed, but can't be changed
- In JDK 7, you can use a **String** to control a **switch** statement:

```
String temp="high";
switch(temp) {
    case "high":
     System.out.println("Switch off the heater");
     break;
    case "low":
     System.out.println("wait");
     break;
```

SubString:

String substring(int start, int end);

To extract substring from index start to end-1

```
System.out.print( “hai hello”.substring(4,9));
// hello
```

Arrays of Strings

```
String strs[] = { "This", "is", "a", "test." };
System.out.println("Original array: ");
for(String s : strs)
   System.out.print(s + " ");
System.out.println("\n"); // change a string
strs[1] = "was";       strs[3] = "test, too!";
System.out.println("Modified array: ");
for(String s : strs)
   System.out.print(s + " ");
```

Output

```
Original array:
This is a test.

Modified array:
This was a test,
too!
```

Command Line Arguments

```
// Display all command-line information.
class CLDemo
 {
        public static void main(String[] args)
        {    System.out.println("There are " + args.length +
                                  " command-line arguments.");
             System.out.println("They are: ");
            for(int i=0; i<args.length; i++)
              System.out.println("args[" + i + "]:  " + args[i]);
        }
}
```

Command Line Arguments

- java CLDemo first second third

- Output:

command line args

There are 3 command-line arguments.
They are:
args[0] : first
args[1] : second
args[2] : third

Question

- Write a program to accept 4 integers from command line and to find and display the average of the numbers.

- Do you find any differences between conventional for loop and the enhanced for loop?

Chapter 6

A Closer Look at Methods and Classes

Introducing Access Control

- Java's access specifiers are **public**, **private**, **default** and **protected**
- **protected** applies only when inheritance is involved
- When a member of a class is modified by the **public** specifier, then that member can be accessed by any other code
- When a member of a class is specified as **private**, then that member can only be accessed by other members of its class

```
/* This program demonstrates the difference
   between public and private. */
class Test {
   int a;                  // default access
   public int b;           // public access
   private int c;          // private access
   // methods to access c
   void setc(int i) {      // set c's value
         c = i;
         }
   int getc() {            // get c's value
         return c;
         }
   }
```

```
class AccessTest {
  public static void main(String args[]) {
        Test ob = new Test();
        // These are OK, a and b may be accessed
  directly
        ob.a = 10;
        ob.b = 20;
        // This is not OK and will cause an error
        //ob.c = 100;                  // Error!
        // You must access c through its methods
        ob.setc(100);                  // OK
        System.out.println("a, b, and c: " + ob.a + " " +
                                              ob.b + " " +
  ob.getc());
        }
  }
```

Passing parameters

- There are two ways that a computer language can pass an argument to a subroutine
- Call by value
- Call by reference

Passing Arguments to Methods

- In Java, arguments are passed using *call-by-value.*
- The value of the argument is copied into the parameter of the method.
- **If the argument is primitive**, then the primitive value is copied and so **changes made to the parameter do not affect the argument**.
- If the **argument is a reference type**, then the reference is copied and so the parameter and **the argument refer to the same object**.

Call by value example:

- When you pass a primitive type to a method, it is passed by value

```
// Simple types are passed by value.
class Test {
  void meth(int i, int j) {
      i *= 2;
      j /= 2;
      }
  }
```

Call by value example:

```
class CallByValue {
  public static void main(String args[]) {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a and b before call: " + a + " "
  + b);
        ob.meth(a, b);
        System.out.println("a and b after call: " + a + " " +
  b);
        }
  }
```

The output from this program is shown here:

```
a and b before call: 15 20
a and b after call: 15 20
```

Call by reference example

- Objects are passed by reference

- Changes to the object inside the method *do* affect the object used as an argument

```
// Objects are passed by reference.
class Test {
        int a, b;

  Test(int i, int j) { a = i;  b = j;}

  // pass an object
  void meth(Test o) {o.a *= 2;  o.b /= 2;}
  }
```

```
class CallByRef {
   public static void main(String args[]) {
         Test ob = new Test(15, 20);
         System.out.println("ob.a and ob.b before call:
   "+ob.a+""+ob.b);

         ob.meth(ob);
         System.out.println("ob.a and ob.b after call: "+ob.a + "
   "+ob.b);
         }
   }
```

This program generates the following output:
ob.a and ob.b before call: 15 20
ob.a and ob.b after call: 30 10

Method Overloading

- Two methods in a class can share a name, as long as they have different parameter declarations.
- Example:

```
class Overload {
  void display() {
    System.out.println("<nothing>");
  }
  void display(int x) {
    System.out.println(x);
  }
}
```

Overloading Methods

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different

- When an overloaded method is invoked, Java uses the **type and/or number of arguments** as its guide to determine which version of the overloaded method to actually call

- The return type alone is insufficient to distinguish two versions of a method

```
class Overload {
  void test() {
      System.out.println("No parameters");}
  // Overload test for one integer parameter.
  void test(int a) {
      System.out.println("a: " + a);}
  // Overload test for two integer parameters.
  void test(int a, int b) {
      System.out.println("a and b: " + a + " " + b);}

  // overload test for a double parameter
  double test(double a) {
      System.out.println("double a: " + a);
      return a*a;}
  }
```

```
class OverloadDemo {

    public static void main(String args[]) {
    Overload ob = new Overload();
    double result;                  // call all versions of test()

    ob.test();
    ob.test(10);
    ob.test(10, 20);
    result = ob.test(123.25);
    System.out.println("Result of ob.test(123.25): " +
                                                    result);
    }
}
```

- In some cases Java's **automatic type conversions** can play a role in overload resolution

- Java will employ its automatic type conversions **only if no exact match is found**

// Automatic type conversions apply to overloading.

```
class Overload2 {
  void test() {
        System.out.println("No parameters");
        }
  // Overload test for two integer parameters.
  void test(int a, int b) {
        System.out.println("a and b: " + a + " " + b);
        }
  // overload test for a double parameter
  void test(double a) {
        System.out.println("Inside test(double) a: " +
  a);
        }
  }
```

```
class OverloadTypeCast {
   public static void main(String args[]) {
         Overload2 ob = new Overload2();
         int i = 88;
         ob.test();
         ob.test(10, 20);
         ob.test(i); // this will invoke test(double)
         ob.test(123.2); //this will invoke test(double)
         }
   }
```

This program generates the following output:

```
No parameters
a and b: 10 20
Inside test(double) a: 88
Inside test(double) a: 123.2
```

Constructor Overloading

- You can overload constructors.
- Example:

```
class Overload{
  int data;
  Overload(int x) {
    data = x;
  }
  Overload(int x, int y) {
    data = x+y;
  }
}
```

Progress Check

1. Name Java's access modifiers
2. What private access specifier does?
3. What is the difference between call-by-value and call-by-reference?
4. How does Java pass primitive types? How does it pass objects?
5. Can a constructor take an object of its own class as a parameter? YES
6. Why might you want to provide overloaded constructors?

Returning Objects

- A method can return any type of data, including class types that you create

```
// Returning an object.
class Test {
   int a;
   Test(int i) {
         a = i;
         }
   Test incrByTen() {
         Test temp = new Test(a+10);
         return temp;
         }
   }
```

```
class RetOb {
   public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after second increase: “ +
   ob2.a);
        }
   }
```

The output generated by this program is shown here:

```
   ob1.a: 2
   ob2.a: 12
   ob2.a after second increase: 22
```

Introducing Nested and Inner Classes

- It is possible to define a class within another class
- The scope of a nested class is bounded by the scope of its enclosing class
- If class B is defined within class A, then B is known to A, but not outside of A
- A nested class has access to the members, including private members, of the class in which it is nested

- However, the enclosing class does not have access to the members of the nested class
- There are two types of nested classes: ***static*** and ***non-static***
- A static nested class is one which has the **static** modifier applied

- The most important type of nested class is the *inner* class
- An inner class is a non-static nested class
- It has access to all of the variables and methods of its outer class

```
// Demonstrate an inner class.
class Outer {
  int outer_x = 100;
  void test() {
       Inner inner = new Inner();
       inner.display();
     }
  // this is an inner class
  class Inner {
       void display() {
       System.out.println("display: outer_x = " + outer_x);
       //100
  }  }    }
class InnerClassDemo {
       public static void main(String args[]) {
                Outer outer = new Outer();
                outer.test();
  }  }
```

- It is important to realize that class **Inner** is known only within the scope of class **Outer**

- The Java compiler generates an error message if any code outside of class **Outer** attempts to instantiate class **Inner**

```
// This program will not compile.
class Outer {
   int outer_x = 100;
   void test() {
   Inner inner = new Inner();   inner.display();
   }
// this is an inner class
   class Inner {
         int y = 10;       // y is local to Inner
         void display() {
                System.out.println("display: outer_x =
                " + outer_x);
         }   }
   void showy() {
   System.out.println(y); // error, y not known
   here!
}    }
```

```
class InnerClassDemo {
 public static void main(String args[]) {
Outer outer = new Outer();
 outer.test();
 }      }
```

- While nested classes are not used in most day-to-day programming, they are particularly helpful when handling events in an applet

Understanding static

- When a member is declared **static**, it can be accessed before any objects of its class are created, and without reference to any object
- The most common example of a **static** member is **main( )**
- **main( )** is declared as **static** because it must be called before any objects exist
- Instance variables declared as **static** are, essentially, global variables

Methods declared as **static** have several restrictions:

- They can only call other **static** methods
- They must only access **static** data
- They cannot refer to **this** in any way
- We can declare a **static** block which gets executed exactly once, when the class is first loaded

```
// Demonstrate static variables, methods, and
   blocks.
class UseStatic {
   static int a = 3;
   static int b;
   static void meth(int x) {
         System.out.println("x = " + x);
         System.out.println("a = " + a);
         System.out.println("b = " + b);
         }
   static {
         System.out.println("Static block
   initialized.");
         b = a * 4;
         }
```

```
    public static void main(String args[ ]) {
        meth(42);
        }
    }
```

```
//output
Static block initialized.
        x = 42
        a = 3
        b = 12
```

- As soon as the **UseStatic** class is loaded, all of the **static** statements are run
- First, **a** is set to **3**, then the **static** block executes (printing a message), and finally, **b** is initialized to **a * 4** or **12**
- Then **main( )** is called, which calls **meth( )**, passing **42** to **x**

- Output of the program :
  static block initialized.
  x=42
  a=3
  b=12

Static count for no. of Instances

```
class MyClass{
static int count=0;
MyClass(){
count++;
}
}
public class StaticCountInstance {

        public static void main(String[] args) {
        for (int i =0; i<3; i++){
              MyClass Ob= new MyClass();
              System.out.println("Object No.: "+ MyClass.count);  }
        }
 }
```

- If you wish to call a **static** method from outside its class, you can do so using the following general form:

classname.method( )

- Here, *classname* is the name of the class in which the **static** method is declared

```
class StaticDemo {
  static int a = 42;
  static int b = 99;
  static void callme() {
      System.out.println("a = " + a);
      }   }
class StaticByName {
  public static void main(String args[]) {
      StaticDemo.callme();
      System.out.println("b = " + StaticDemo.b);
      }   }
```

Here is the output of this program:

```
a = 42
b = 99
```

Static Inner Class

```
1.class TestOuter1{
2.  static int data=30;
3.  static class Inner{
4.   void msg(){System.out.println("data is "+data);}
5.  }
6.  public static void main(String args[]){
7.  TestOuter1.Inner obj=new TestOuter1.Inner();
8.  obj.msg();
9.  }
10.}
```

Introducing final

- A variable can be declared as **final**
- Doing so prevents its contents from being modified
- We must initialize a **final** variable when it is declared
- final int FILE_NEW = 1;
- final int FILE_OPEN = 2;

- Variables declared as **final** do not occupy memory on a per-instance basis

- The keyword **final** can also be applied to methods, but its meaning is substantially different than when it is applied to variables

Inner / Nested Classes

- A class declared within another class is called an *inner class*.
- Its scope is the enclosing class.
- It has access to all the variables and methods of the enclosing class.

Example of an Inner Class

```
class Outer {
  int x = 5;

  class Inner {
    void changeX() { x = 3; } // change Outer's x
  }

  void adjust() {
    Inner inn = new Inner();
    inn.changeX(); // Outer's x is now 3
  }
}
```

Varargs

- Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments. This feature is called *varargs* and it is short for *variable-length arguments*.
- A method that takes a variable number of arguments is called a *variable-arity method*, or simply a *varargs method*.

Syntax of varargs:

- The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

return_type method_name(data_type... variableName){}

Varargs

- You can create methods with a variable number of arguments, using "..."
- Example:

```
class VATest {
   static void vaMethod(int ... data) {
     for(int x : data) System.out.print(x + " ");
   }
   public static void main(String[] args) {
     vaMethod(1); // prints "1 "
     vaMethod(1,2,3); // prints "1 2 3 "
   }
}
```

Overloading Varargs Methods

- There must be only one varargs parameter.
- You can overload a method that takes a variable-length argument.

```
static void vaTest(int ... v) {
static void vaTest(boolean ... v) {
static void vaTest(String msg, int ... v) {
```

- *A varargs method can also be overloaded by a non-varargs method.*
- unexpected errors can result when overloading a method that takes a variable-length argument. These errors involve ambiguity because it is possible to create an ambiguous call to an overloaded varargs method.

Overloading Varargs Methods

- Ambiguity can result in some cases.
- Example:

```
class VAOverload {
  static void vaMethod(int ... data) { }
  static void vaMethod(boolean ... data) { }
}
```

- This is fine unless you call **vaMethod( )** with no arguments, which is ambiguous.

Exercise

Q1. Suppose you have a class MyClass with one instance variable x. What will be printed by the following code segment? Explain your answer.

```
MyClass c1 = new MyClass();
c1.x = 3;
MyClass c2 = c1;
c2.x = 4;
System.out.println(c1.x);
```

Exercise

Q2. Suppose that a class has an overloaded method named add with the following two implementations:

```
double add(int x, double y) { return x + y; }
double add(double x, int y) { return x + y + 1; }
```

What, if anything will be returned by the following method calls?

A. add(3, 3.14)
B. add(3.14, 3)
C. add(3, 3)
D. add(3.14, 3.14)