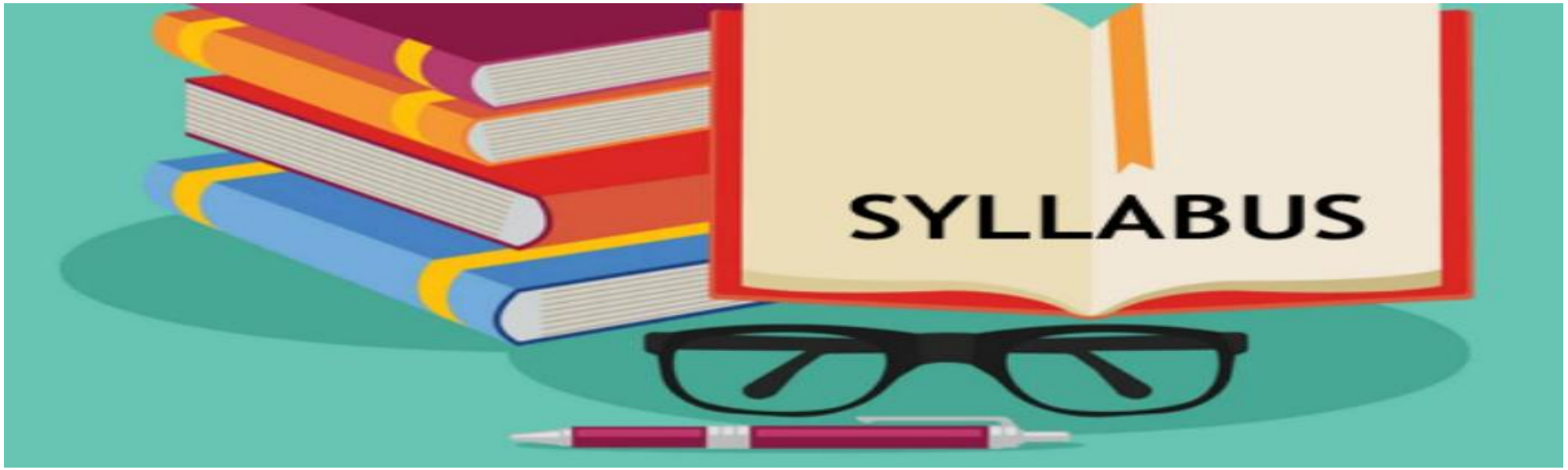
SYLLABUS OF SEMESTER – I

Course : MCA Course: Object oriented programming



Course Objectives

The objective of the course is to prepare the students:

1. To enable the development of skills through which the student will gain expertise in writing programs using object oriented programming features.
2. Learn to apply concepts of File handling, exception handling.
3. To develop various programs on Generics, Collections and multithreading

Course Outcomes

After the completion of the course, students will be able to

CO1: Discuss and analysis of different features object oriented programming.

CO2: Develop basic programs for given problems.

CO3: Discuss the File handling and exception handling and develop programs using concept

Of error handling

CO4: Discuss Generics, Collections and multithreading and develop programs using these concepts.

Syllabus

Module-I (Teaching Hours – 7)

Features of Object Oriented Programming languages like data encapsulation, inheritance, polymorphism and late binding. Introduction to class and Methods, Access control of members of a class, instantiating a class, Constructors, Garbage Collection, finalize() Method.

Module-II (Teaching Hours – 7)

Concept of inheritance, methods of derivation, use of super keyword and final keyword in inheritance, run time polymorphism. Abstract classes and methods, interface, implementation of interface, creating packages, importing packages, static and non-static members.

Module-III (Teaching Hours - 8)

Exceptions, types of exception, use of try catch block, handling multiple exceptions, using finally, throw and throws clause, user defined exceptions, Generics, generic class with two type parameter, bounded generics, Collection classes: Arrays, Vectors, Array list, Linked list, Hash set, Queues, Trees.

Module-IV (Teaching Hours - 8)

Introduction to streams, byte streams, character streams, file handling in Java, Serialization Multithreading: Java Thread models, creating thread using runnable interface and extending Thread, thread priorities, Thread Synchronization, Inter-thread communications.

Module - V :

JSP-Why JSP?, JSP Directives, Writing simple JSP page, Scripting Elements, Default Objects in JSP, JSP Actions, Managing Sessions using JSP, JSP with beans. Java Database Connectivity, Servlets - Introduction Servlets vs CGI, Servlets API Overview, Servlets Life Cycle, Coding Writing & runningsimple Servlets, Generic Servlets, HTTPServlet, Servlets Config, Servlets Contest Writing Servlets to handle Get& Post methods.

Text Books:

1. **JAVA The Complete Reference: Herbert Schildt; Seventh Edition, Tata McGraw- Hill Publishing Company Limited 2007.**

2. A programmer's Guide to Java SCJP Certification: A Comprehensive Primer: Khalid A. Mughal and Rolf W.Rasmussen, Third Edition.

3. Java Fundamentals: A Comprehensive Introduction:HerbertSchildt and Dale Skrien; Tata McGraw- Hill Education Private Ltd., 2013.

Reference Books:

1. Core JAVA Volume-II Advanced Features: Cay S. Horstmann and Gary Cornell; Eighth Edition; Prentice Hall, Sun Microsystems Press, 2008.

2. Java Programming: A Practical Approach: C Xavier; Tata McGraw- Hill Education Private Ltd., 2011.

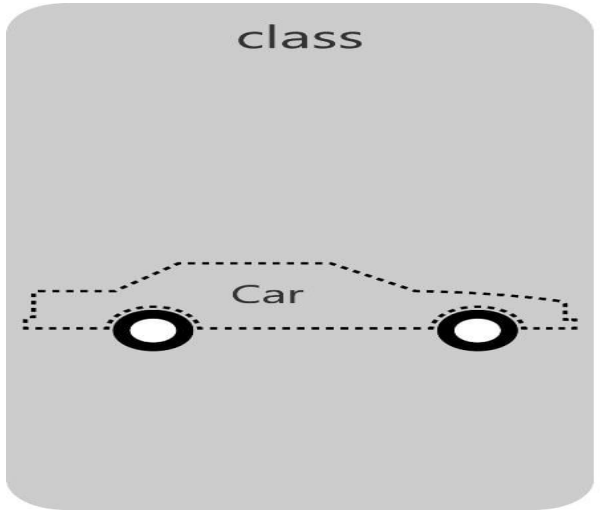
Module-I (Teaching Hours – 7)

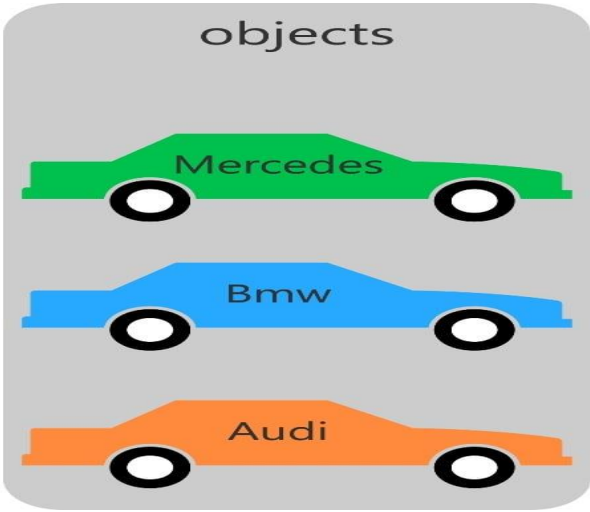
Features of Object Oriented Programming languages like data encapsulation, inheritance, polymorphism and late binding. Introduction to class and Methods, Access control of members of a class, instantiating a class, Constructors, Garbage Collection, finalize() Method.

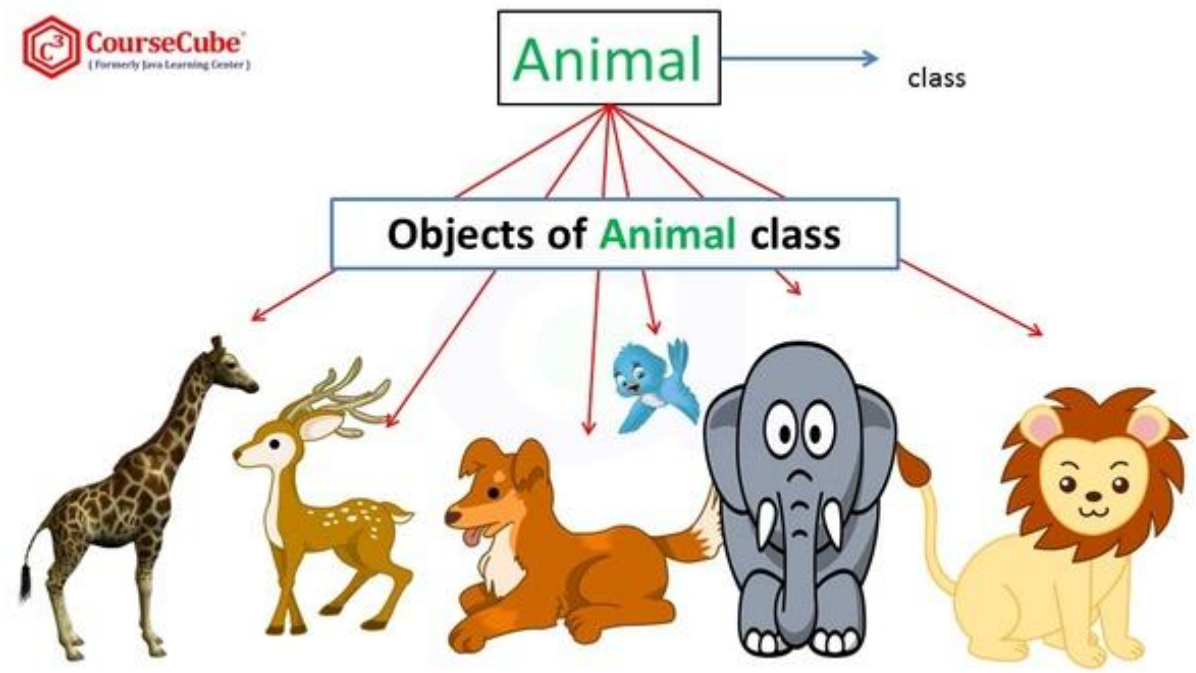
Java was conceived by James Gosling, Patrick Naughton, Chris Warth, Ed Frank, and Mike Sheridan at Sun Microsystems, Inc. in 1991. It took 18 months to develop the first working version. This language was initially called “Oak,” but was renamed “Java” in 1995.

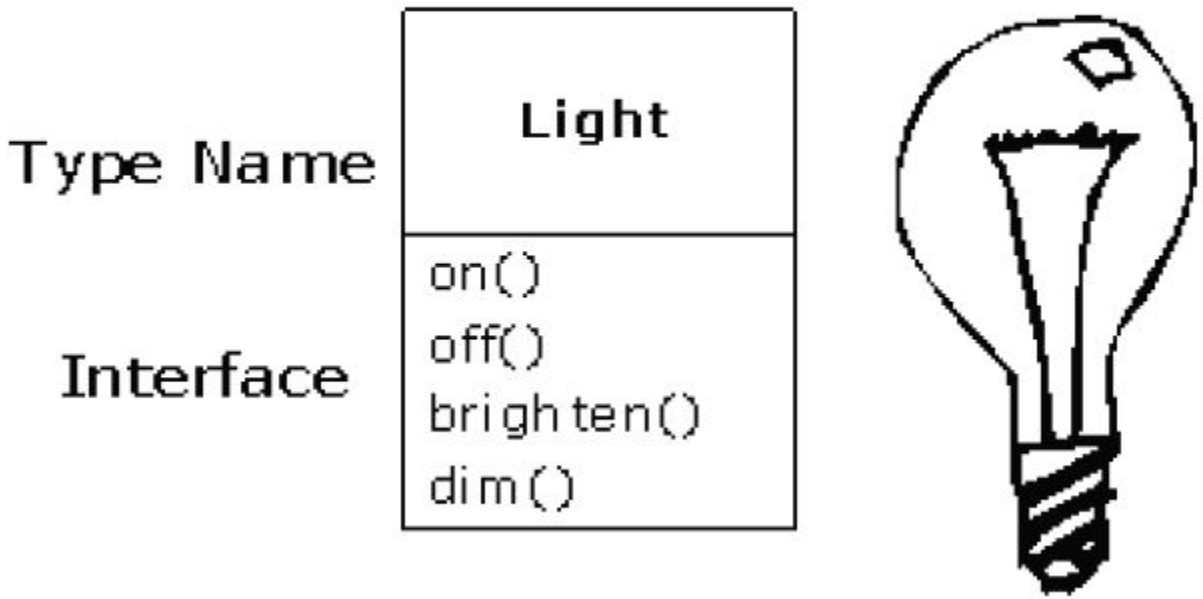
Main Concepts

- Object
- Class
- Inheritance
- Encapsulation
- Abstraction
- Polymorphism





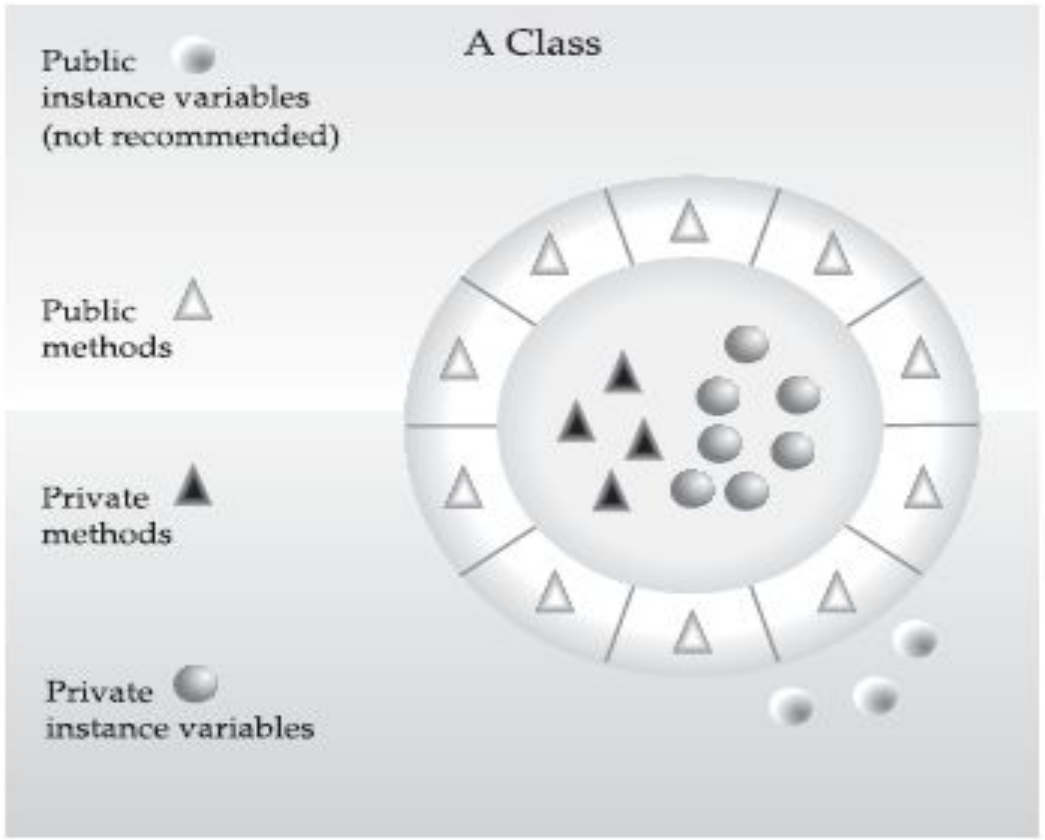




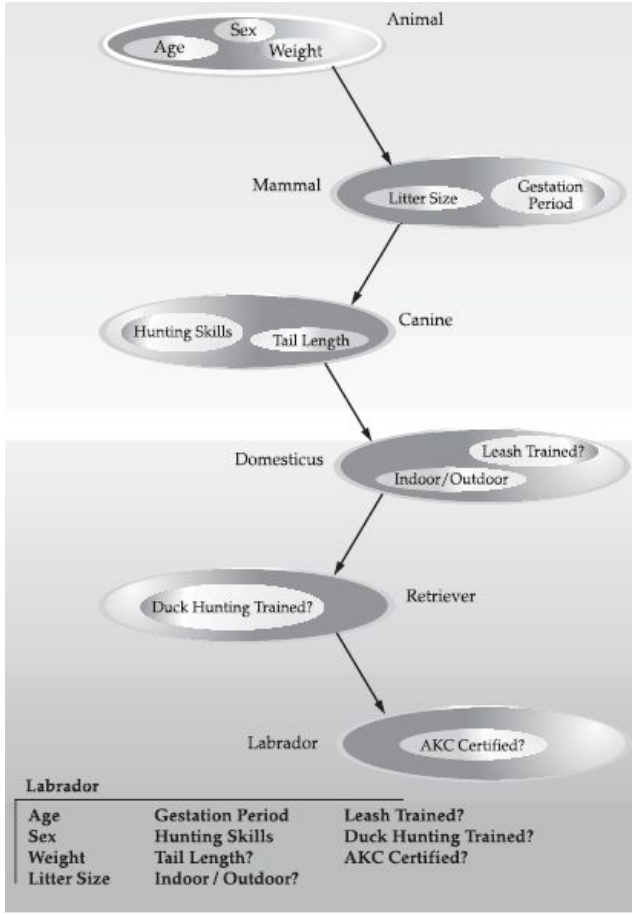
An object has state, behavior and identity.

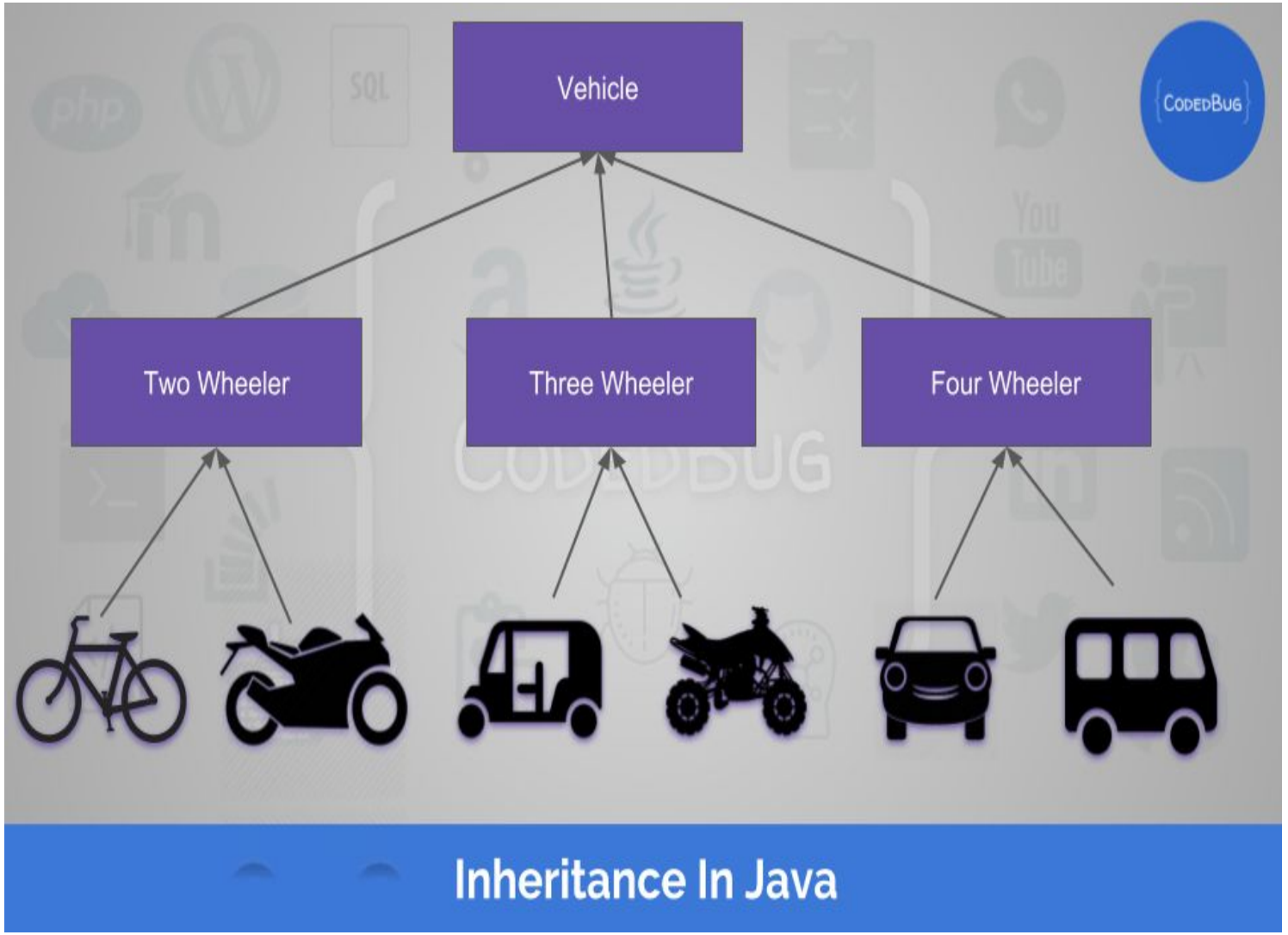
encapsulation

FIGURE 2-1
Encapsulation: public methods can be used to protect private data



inheritance

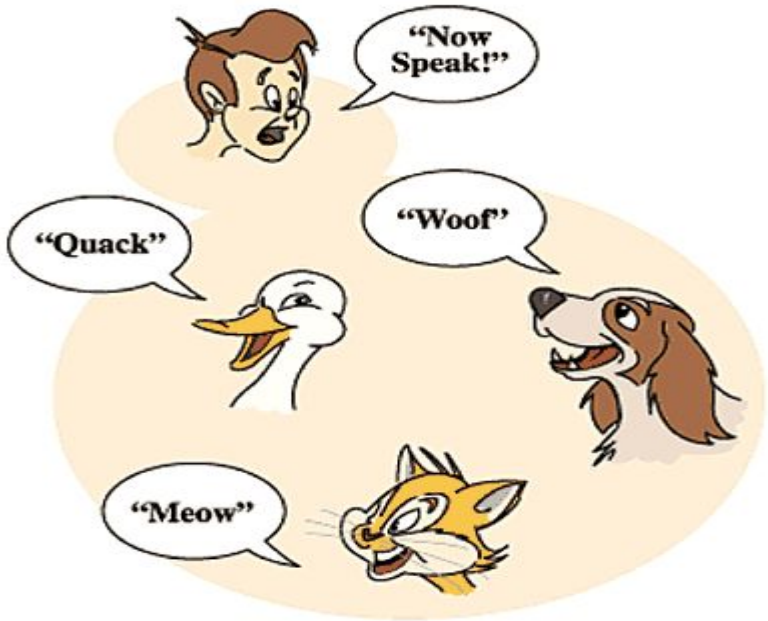




Polymorphism

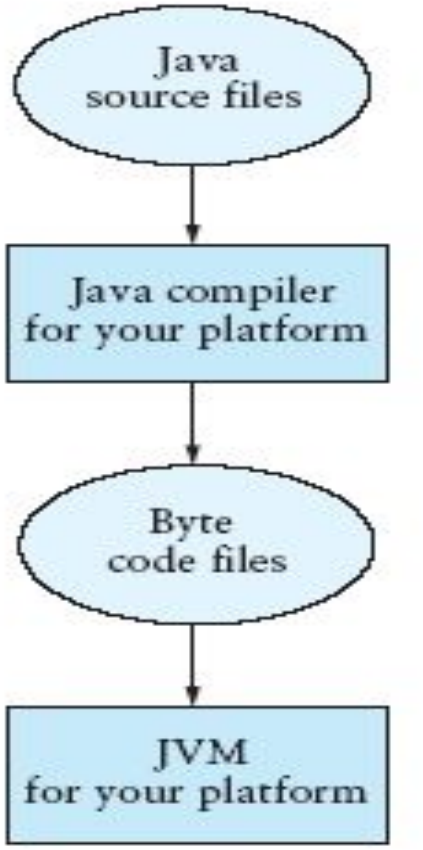
Extending the dog analogy, a dog's sense of smell is polymorphic.

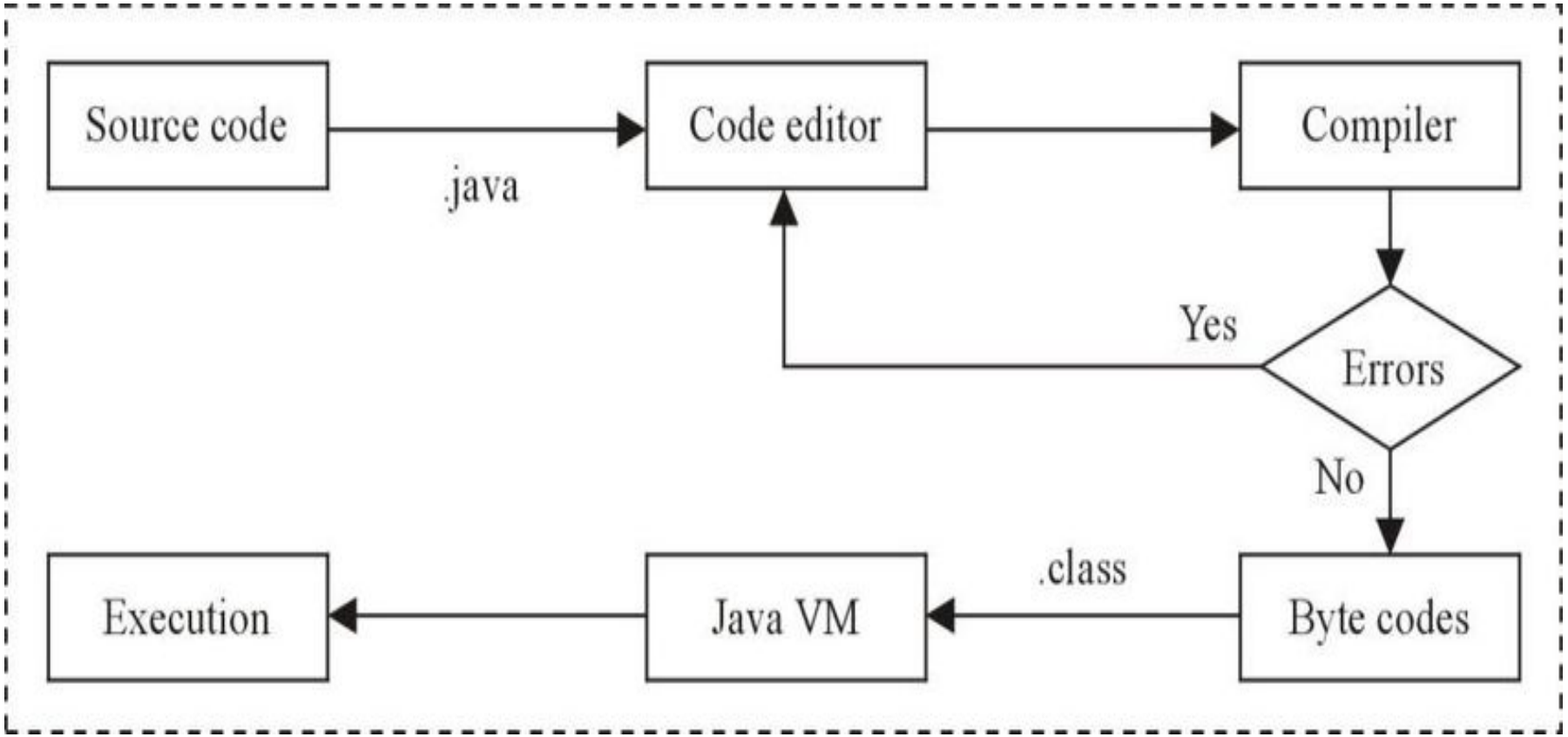
If the dog smells a cat,it will bark and run after it. If the dog smells its food, it will salivate and run to its bowl.
The same sense of smell is at work in both situations. The difference is what is being smelled, that is, the type of data being operated upon by the dog's nose! This same general concept can be implemented in Java as it applies to methods within a Java program.



Compiling and Executing a Java Program

FIGURE A.1
Compiling and Executing a Java Program





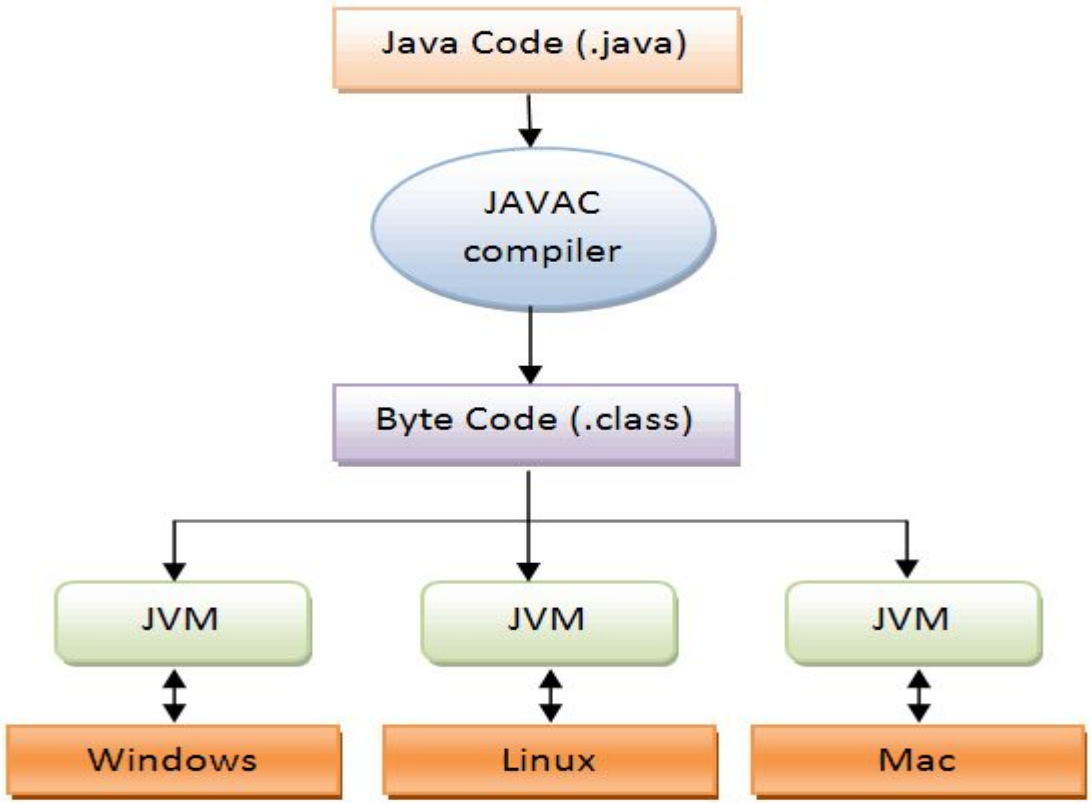
Java's Magic: The Bytecode

The key that allows Java to solve both the security and the portability problems just described is that 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).*** In essence, the original JVM was designed as an ***interpreter for bytecode***. This may come as a bit of a surprise since many modern languages are designed to be compiled into executable code because of performance concerns. However, the fact that a Java program is executed by the JVM helps solve the major problems associated with web-based programs. **Here is why.**

Translating a Java program into bytecode makes it much easier to run a program in a wide variety of environments because only the JVM needs to be implemented for each platform. Once the run-time package exists for a given system, any Java program can run on it. Remember, although the details of the JVM will differ from platform to platform, all understand the same Java bytecode. If a Java program were compiled to native code, then different versions of the same program would have to exist for each type of CPU connected to the Internet. This is, of course, not a feasible solution. Thus, the execution of bytecode by the JVM is the easiest way to create truly portable programs.

The fact that a Java program is executed by the JVM also helps to make it secure. Because the JVM is in control, it can contain the program and prevent it from generating side effects outside of the system. As you will see, safety is also enhanced by certain restrictions that exist in the Java language.

In general, when a program is compiled to an intermediate form and then interpreted by a virtual machine, it runs slower than it would run if compiled to executable code.
However, with Java, the differential between the two is not so great. Because bytecode has been highly optimized, the use of bytecode enables the JVM to execute programs much faster than you might expect.



```
/*
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.");
      }
}
```

The first thing that you must learn about Java is that the name you give to a source file is very important. For this example, the name of the source file should be **Example.java**.

To compile the **Example** program, execute the compiler, **javac**, specifying the name of the source file on the command line, as shown here:

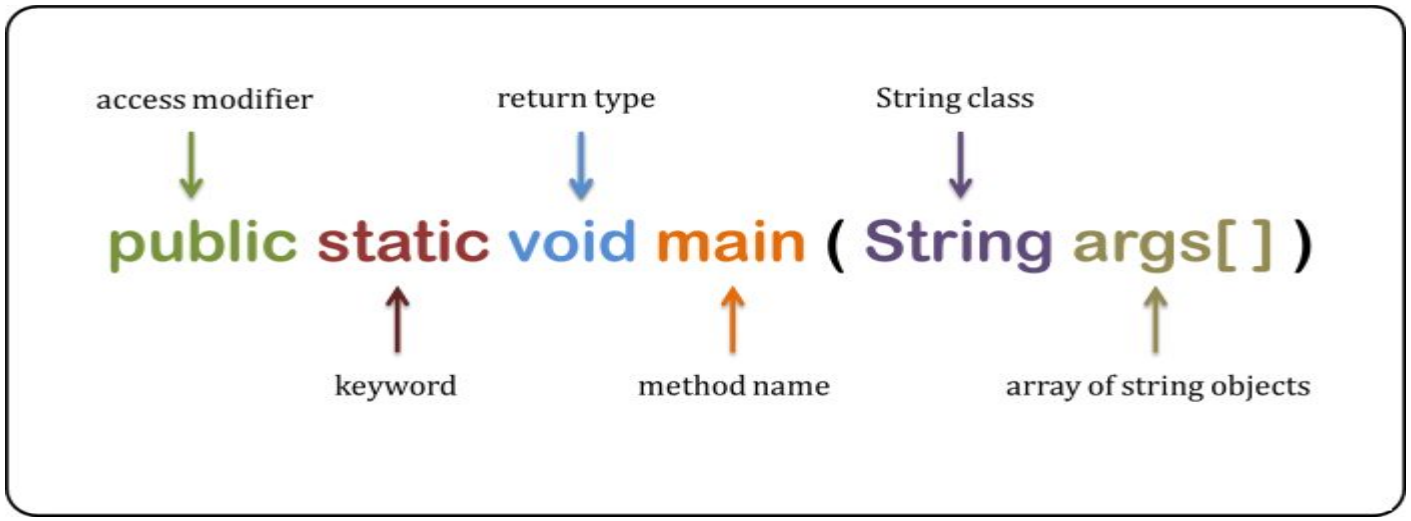
C:\>**javac Example.java**

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

To actually run the program, you must use the Java application launcher, called **java**.
To do so, pass the class name **Example** as a command-line argument, as shown here:

C:\>**java Example**

When the program is run, the following output is displayed:
This is a simple Java program.



This line begins the **main( )** method. As the comment preceding it suggests, this is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**.

The **public** keyword is an *access specifier*, which allows the programmer to control the visibility of class members. 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 opposite of **public** is **private**, which prevents a member from being used by code defined outside of its class.)
In this case, **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started. The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java Virtual Machine before any objects are made. The keyword **void** simply tells the compiler that **main( )** does not return a value.

As stated, **main( )** is the method called when a Java application begins. Keep in mind that **Java is case-sensitive**. Thus, **Main** is different from **main**. It is important to understand that the **Java compiler will compile classes that do not contain a main( ) method**. **But java has no way to run these classes.** So, if you had typed **Main** instead of **main**, the compiler would still compile your program. However, **java** would report an error because it would be unable to find the **main( )** method.

```
System.out.println("This is a simple Java program.");
```

System is a predefined class that provides access to the system, and **out** is the output stream that is connected to the console.

println( ) displays the string which is passed to it.

```
class CommandLineExample
{
	public static void main(String args[])
	{
		System.out.println("Your first argument is: "+args[0]);
	}
}
```

compile by > javac CommandLineExample.java
run by > java CommandLineExample sonoo

Output: Your first argument is: sonoo

Another example

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

compile by > javac Sample.java
run by > java Sample 10 20 30

Output:

```
10
20
30
```

```
/*
Here is another short example.
Call this file "Example2.java".
*/
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);
	}
}
```

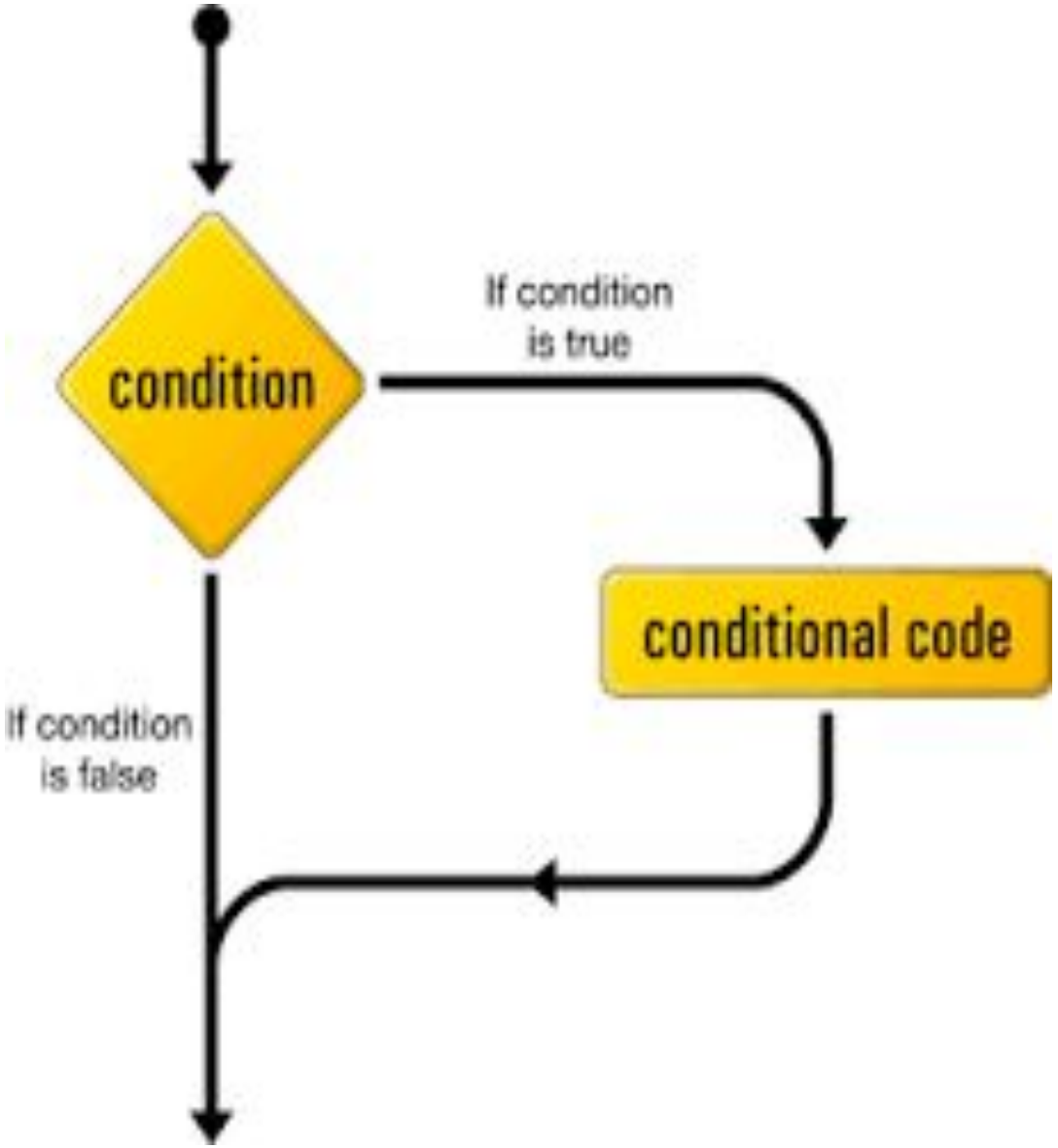
When you run this program, you will see the following output:

```
This is num: 100
The value of num * 2 is 200
```

```
/*
Demonstrate the if.
Call this file "IfSample.java".
*/
class IfSample {
public static void main(String args[]) {
int x, y;
x = 10;
y = 20;
if(x < y) System.out.println("x is less than y");
x = x * 2;
if(x == y) System.out.println("x now equal to y");
x = x * 2;
if(x > y) System.out.println("x now greater than y");
// this won't display anything
if(x == y) System.out.println("you won't see this");
}
}
```

The output generated by this program is shown here:

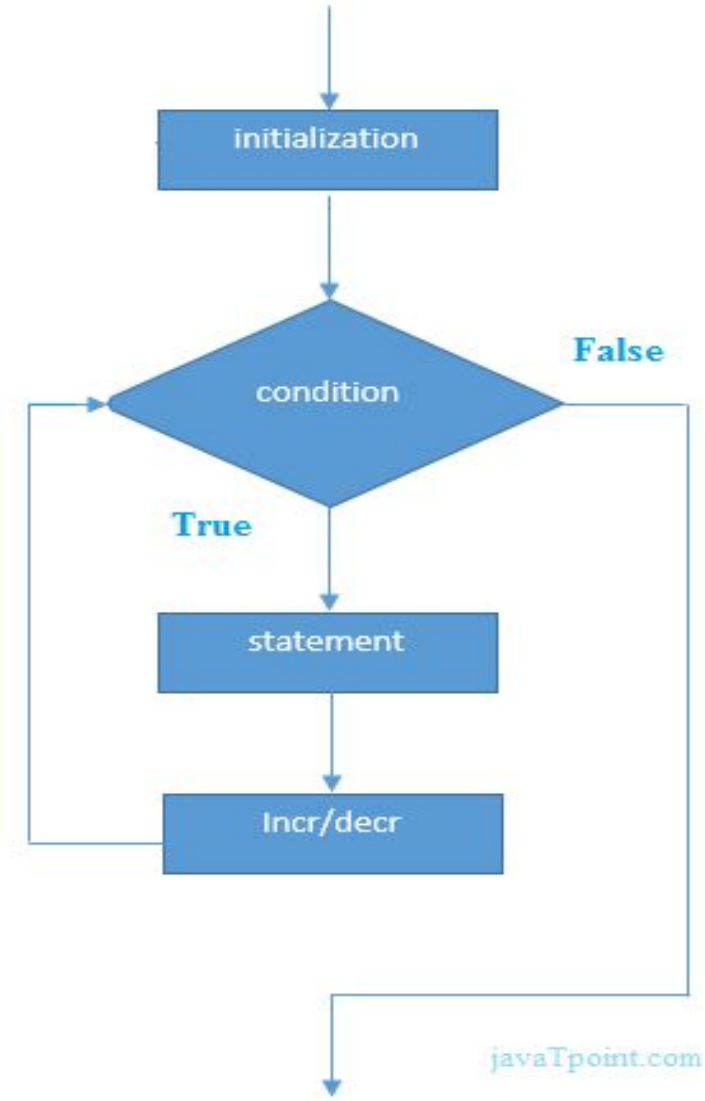
```
x is less than y
x now equal to y
x now greater than y
```



```
/*
Demonstrate the for loop.
Call this file "ForTest.java".
*/
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);
}
}
```

This program generates the following output:

```
This is x: 0
This is x: 1
This is x: 2
This is x: 3
This is x: 4
This is x: 5
This is x: 6
This is x: 7
This is x: 8
This is x: 9
```



```
// Demonstrate block scope.
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);
	}
}
```

```
// Demonstrate a one-dimensional array.
class Array
{
	public static void main(String args[])
	{
	int month_days[];
	month_days = new int[12];
	month_days[0] = 31;
	month_days[1] = 28;
	month_days[2] = 31;
	month_days[3] = 30;
	month_days[4] = 31;
	month_days[5] = 30;
	month_days[6] = 31;
	month_days[7] = 31;
	month_days[8] = 30;
	month_days[9] = 31;
	month_days[10] = 30;
	month_days[11] = 31;
	System.out.println("April has " + month_days[3] + " days.");
	}
}
```

```
// An improved version of the previous program.

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.");
	}
}
```

```
// Average an array of values.
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);
	}
}
```

```
String str = “this is test string”;
System.out.println(str);
```

Classes and objects

A Simple Class

Let's begin our study of the class with a simple example. Here is a class called **Box** that defines three instance variables: **width**, **height**, and **depth**. Currently, **Box** does not contain any methods (but some will be added soon).

```
class Box {
double width;
double height;
double depth;
}
```

A class defines a new type of data. In this case, the new data type is called **Box**. You will use this name to declare objects of type **Box**. It is important to remember that a **class** declaration only creates a template; it does not create an actual object. Thus, the preceding code does not cause any objects of type **Box** to come into existence.

To actually create a **Box** object, you will use a statement like the following:

Box mybox = new Box(); // create a Box object called mybox

After this statement executes, **mybox** will be an instance of **Box**. Thus, it will have "physical" reality.

To assign the **width** variable of **mybox** the value 100, you would use the following statement:

```
mybox.width = 100;
```

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
        double width;
        double height;
        double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
        Box mybox = new Box();
        double vol;
        // assign values to mybox's instance variables
        mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
// compute volume of box
                vol = mybox.width * mybox.height *
                mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

You should call the file that contains this program **BoxDemo.java**, because the **main( )** method is in the class called **BoxDemo**, not the class called **Box**. When you compile this program, you will find that two **.class** files have been created, one for **Box** and one for **BoxDemo**. The Java compiler automatically puts each class into its own **.class** file. It is not necessary for both the **Box** and the **BoxDemo** class to actually be in the same source file. You could put each class in its own file, called **Box.java** and **BoxDemo.java**, respectively.

To run this program, you must execute **BoxDemo.class**. When you do, you will see the
following output:
Volume is 3000.0

```
// This program declares two Box objects.

class Box {
        double width;
        double height;
        double depth;
}
class BoxDemo2 {
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // assign values to mybox1's instance variables
        mybox1.width = 10;
        mybox1.height = 20;
        mybox1.depth = 15;
        /* assign different values to mybox2's
        instance variables */
        mybox2.width = 3;
        mybox2.height = 6;
        mybox2.depth = 9;
        // compute volume of first box
        vol = mybox1.width * mybox1.height * mybox1.depth;
        System.out.println("Volume is " + vol);
        // compute volume of second box
        vol = mybox2.width * mybox2.height * mybox2.depth;
        System.out.println("Volume is " + vol);
        }
}
```

The output produced by this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

As you can see, **mybox1**'s data is completely separate from the data contained in **mybox2**.

when you create a class, you are creating a new data type. You can use this type to declare objects of that type. However, obtaining objects of a class is a two-step process. 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 (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by **new**. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated. Let's look at the details of this procedure.
In the preceding sample programs, a line similar to the following is used to declare an object of type **Box**:

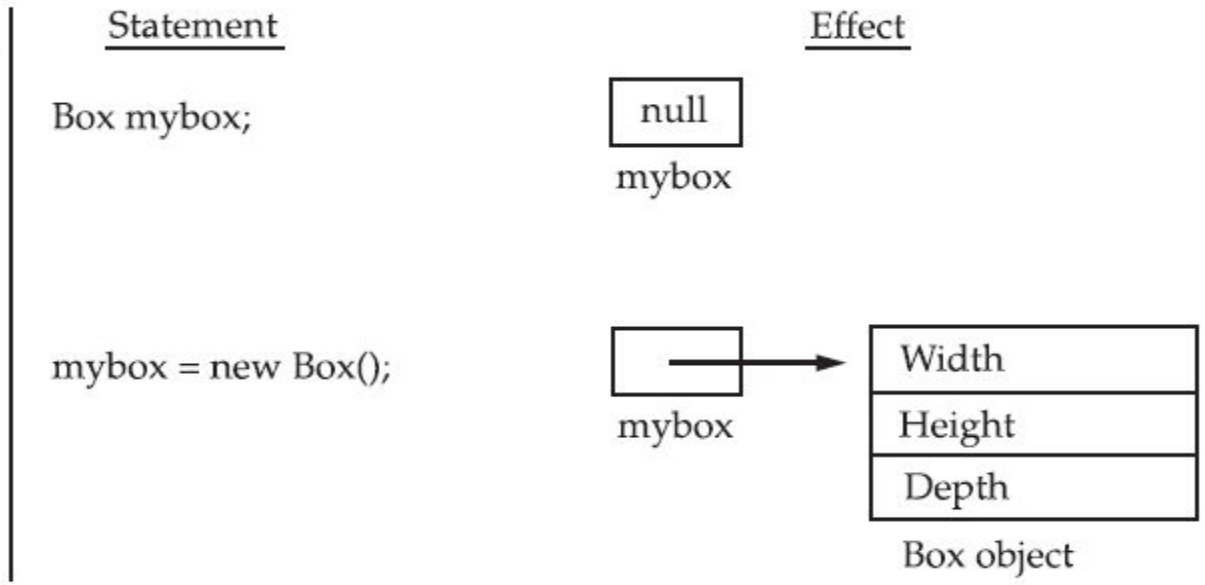
```
Box mybox = new Box();
```

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

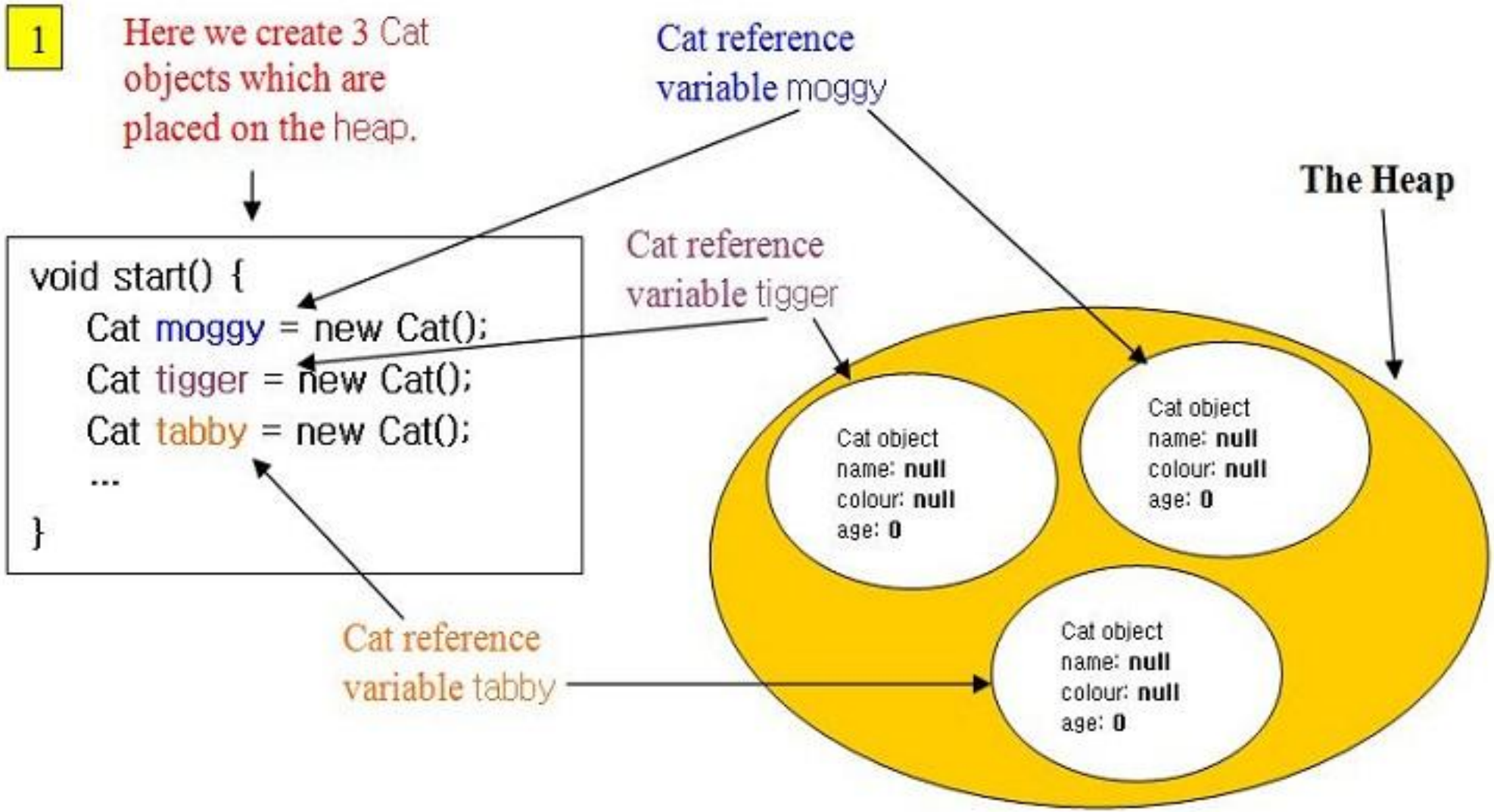
```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares **mybox** as a reference to an object of type **Box**. After this line executes, **mybox** contains the value **null**, which indicates that it does not yet point to an actual object. Any attempt to use **mybox** at this point will result in a compile-time error. The next line allocates an actual object and assigns a reference to it to **mybox**. After the second line executes, you can use **mybox** as if it were a **Box** object. But in reality, **mybox** simply holds the memory address of the actual **Box** object.

FIGURE 6-1
Declaring an object of type **Box**



the distinction between a class and an object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When you declare an object of a class, you are creating an instance of that class. Thus, a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.) It is important to keep this distinction clearly in mind.


1
Here we create 3 Cat objects which are placed on the heap.
Cat reference variable moggy
The Heap
void start() {
Cat moggy = new Cat();
Cat tigger = new Cat();
Cat tabby = new Cat();
...
}
Cat reference variable tigger
Cat object
name: null
colour: null
age: 0
Cat object
name: null
colour: null
age: 0
Cat object
name: null
colour: null
age: 0
Cat reference variable tabby

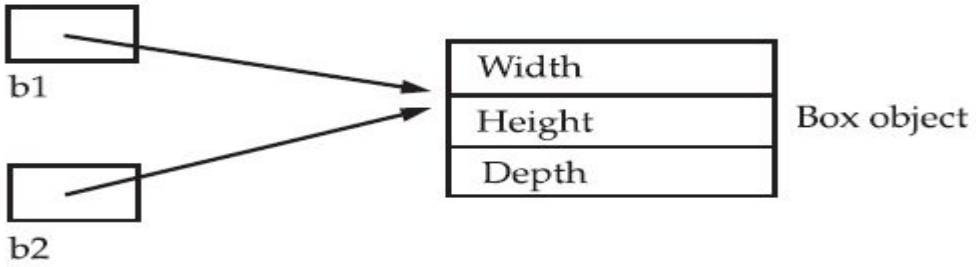
Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

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

You might think that **b2** is being assigned a reference to a copy of the object referred to by **b1**. That is, you might think that **b1** and **b2** refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, **b1** and **b2** will both refer to the *same* object. The assignment of **b1** to **b2** did not allocate any memory or copy any part of the original object. It simply makes **b2** refer to the same object as does **b1**. Thus, any changes made to the object through **b2** will affect the object to which **b1** is referring, since they are the same object.

This situation is depicted here:



Although **b1** and **b2** both refer to the same object, they are not linked in any other way. For example, a subsequent assignment to **b1** will simply *unhook* **b1** from the original object without affecting the object or affecting **b2**. For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, **b1** has been set to **null**, but **b2** still points to the original object.

// This program includes a method inside the box class.

```
class Box
{
	double width;
	double height;
	double depth;
	// display volume of a box
	void volume()
	{
		System.out.print("Volume is ");
		System.out.println(width * height * depth);
	}
}
```

This program generates the following output, which is the same as the previous version.

```
Volume is 3000.0
Volume is 162.0
```

```
class BoxDemo3
{
public static void main(String args[])
{
	Box mybox1 = new Box();
	Box mybox2 = new Box();
	// assign values to mybox1's instance variables
	mybox1.width = 10;
	mybox1.height = 20;
	mybox1.depth = 15;
	/* assign different values to mybox2's
	instance variables */
	mybox2.width = 3;
	mybox2.height = 6;
	mybox2.depth = 9;
	// display volume of first box
	mybox1.volume();
	// display volume of second box
	mybox2.volume();
	}
}
```

```
// Now, volume() returns the volume of a box.
class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
        double volume()
        {
                return width * height * depth;
        }
}
class BoxDemo4
{
        public static void main(String args[])
        {
                Box mybox1 = new Box();
                Box mybox2 = new Box();
                double vol;
                // assign values to mybox1's instance var.
                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;
                /* assign different values to mybox2's
                instance variables */
                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;
                // get volume of first box
                vol = mybox1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box
                vol = mybox2.volume();
                System.out.println("Volume is " + vol);
                }
        }
```

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is **boolean**, you could not return an integer.
- The variable receiving the value returned by a method (such as **vol**, in this case) must also be compatible with the return type specified for the method.

System.out.println("Volume is " + mybox1.volume());
when **println( )** is executed, **mybox1.volume( )** will be called automatically and its value will be passed to **println( )**.

```
// This program uses a parameterized method.
class Box {
        double width;
        double height;
        double depth;
        // compute and return volume
        double volume()
        {
        return width * height * depth;
        }
        // sets dimensions of box
        void setDim(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
}
```

```
class BoxDemo5 {
public static void main(String args[]) {
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // initialize each box
        mybox1.setDim(10, 20, 15);
        mybox2.setDim(3, 6, 9);
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

the **setDim( )** method is used to set the dimensions of each box. For example, when

mybox1.setDim(10, 20, 15);

is executed, 10 is copied into parameter **w**, 20 is copied into **h**, and 15 is copied into **d**. Inside **setDim( )** the values of **w**, **h**, and **d** are then assigned to **width**, **height**, and **depth**, respectively.

```
/* Here, Box uses a constructor to initialize the
dimensions of a box.
*/
class Box
{
        double width;
        double height;
        double depth;
        // This is the constructor for Box.
        Box()
        {
                System.out.println("Constructing Box");
                width = 10;
                height = 10;
                depth = 10;
         }
        // compute and return volume
        double volume()
        {
                return width * height * depth;
        }
}
```

```
class BoxDemo6 {
public static void main(String args[])
 {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box();
        Box mybox2 = new Box();
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

When this program is run, it generates the following results:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

```
/* Here, Box uses a parameterized constructor to
initialize the dimensions of a box.
*/
class Box {
        double width;
        double height;
        double depth;
        // This is the constructor for Box.
        Box(double w, double h, double d)
        {
        width = w;
        height = h;
        depth = d;
        }
        // compute and return volume
        double volume() {
        return width * height * depth;
        }
}
```

```
class BoxDemo7 {
public static void main(String args[]) {
        // declare, allocate, and initialize Box objects
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box(3, 6, 9);
        double vol;
        // get volume of first box
        vol = mybox1.volume();
        System.out.println("Volume is " + vol);
        // get volume of second box
        vol = mybox2.volume();
        System.out.println("Volume is " + vol);
        }
}
```

The output from this program is shown here:

```
Volume is 3000.0
Volume is 162.0
```

Introducing Access Control

Java's access specifiers are **public**, **private**, and **protected**. Java also defines a default access level. **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.
Now you can understand why **main( )** has always been preceded by the **public** specifier. It is called by code that is outside the program—that is, by the Java run-time system. When no access specifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package. (Packages are discussed in the following chapter.)
In the classes developed so far, all members of a class have used the default access mode, which is essentially public. However, this is not what you will typically want to be the case.
Usually, you will want to restrict access to the data members of a class—allowing access only through methods. Also, there will be times when you will want to define methods that are private to a class.
An access specifier precedes the rest of a member's type specification. That is, it must begin a member's declaration statement.
Here is an example:

```
public int i;
private double j;
private int myMethod(int a, char b) { // ...
```

```
/* 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());
        }
}
```

As you can see, inside the **Test** class, **a** uses default access, which for this example is the same as specifying **public**. **b** is explicitly specified as **public**. Member **c** is given private access. This means that it cannot be accessed by code outside of its class. So, inside the **AccessTest** class, **c** cannot be used directly. It must be accessed through its public methods: **setc( )** and **getc( )**. If you were to remove the comment symbol from the beginning of the following line,

// ob.c = 100; // Error!

then you would not be able to compile this program because of the access violation.

Introducing final

A variable can be declared as **final**. Doing so prevents its contents from being modified.
This means that you must initialize a **final** variable when it is declared. For example:
final int FILE_NEW = 1;

```
// Demonstrating Strings.
class StringDemo {
public static void main(String args[]) {
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1 + " and " + strOb2;
        System.out.println(strOb1);
        System.out.println(strOb2);
        System.out.println(strOb3);
        }
}
```

The output produced by this program is shown here:

```
First String
Second String
First String and Second String
```

```
// Demonstrating some String methods.
class StringDemo2 {
public static void main(String args[])
{
        String strOb1 = "First String";
        String strOb2 = "Second String";
        String strOb3 = strOb1;
        System.out.println("Length of strOb1: " +
        strOb1.length());
        System.out.println("Char at index 3 in strOb1: " +
        strOb1.charAt(3));
        if(strOb1.equals(strOb2))
        System.out.println("strOb1 == strOb2");
        else
        System.out.println("strOb1 != strOb2");
        if(strOb1.equals(strOb3))
        System.out.println("strOb1 == strOb3");
        Else
        System.out.println("strOb1 != strOb3");
        }
}
```

This program generates the following output:

```
Length of strOb1: 12
Char at index 3 in strOb1: s
strOb1 != strOb2
strOb1 == strOb3
```

Of course, you can have arrays of strings, just like you can have arrays of any other type of object. For example:

```
// Demonstrate String arrays.
class StringDemo3 {
public static void main(String args[]) {
        String str[] = { "one", "two", "three" };
        for(int i=0; i<str.length; i++)
        System.out.println("str[" + i + "]: " +
        str[i]);
        }
}
```

Here is the output from this program:

```
str[0]: one
str[1]: two
str[2]: three
```

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the **this** keyword. **this** can be used inside any method to refer to the *current* object. That is, **this** is always a reference to the object on which the method was invoked. You can use **this** anywhere a reference to an object of the current class' type is permitted.

To better understand what **this** refers to, consider the following version of **Box( )**:

```
// A redundant use of this.

Box(double w, double h, double d) {
this.width = w;
this.height = h;
this.depth = d;
}
```

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 this is the case, the methods are said to be *overloaded,* and the process is referred to as *method overloading.* Method overloading is one of the ways that Java supports polymorphism.
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. Thus, overloaded methods must differ in the type and/or number of their parameters.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

```
// Demonstrate method overloading.
class OverloadDemo {
        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 Overload {
public static void main(String args[]) {
        OverloadDemo ob = new OverloadDemo();
        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);
        }
}
```

This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
class Box {
double width;
double height;
double depth;
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
        }
}
class OverloadCons {
public static void main(String args[]) {
    // create boxes using the various constructors
    Box mybox1 = new Box(10, 20, 15);
    Box mybox2 = new Box();
    Box mycube = new Box(7);
    double vol;
    // get volume of first box
    vol = mybox1.volume();
    System.out.println("Volume of mybox1 is " + vol);
    // get volume of second box
    vol = mybox2.volume();
    System.out.println("Volume of mybox2 is " + vol);
    // get volume of cube
    vol = mycube.volume();
    System.out.println("Volume of mycube is " + vol);
    }
}
```

The output produced by this program is shown here:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

Using Objects as Parameters

So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program:

```
// Objects may be passed to methods.
class Test
{
	int a, b;
	Test(int i, int j)
	{
		a = i;
		b = j;
	}
	// return true if o is equal to the invoking object
	boolean equals(Test o)
	{
		if(o.a == a && o.b == b) return true;
		else
		return false;
	}
}
```

```
class PassOb
{
	public static void main(String args[])
	{
	Test ob1 = new Test(100, 22);
	Test ob2 = new Test(100, 22);
	Test ob3 = new Test(-1, -1);
	System.out.println("ob1 == ob2: " + ob1.equals(ob2));

	System.out.println("ob1 == ob3: " + ob1.equals(ob3));
	}
}
```

This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

```
// Here, Box allows one object to initialize another.
class Box {
double width;
double height;
double depth;
// Notice this constructor. It takes an object of type Box.
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons2 {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7);
Box myclone = new Box(mybox1); // create copy of mybox1
double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box
vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube
vol = mycube.volume();
System.out.println("Volume of cube is " + vol);
// get volume of clone
vol = myclone.volume();
System.out.println("Volume of clone is " + vol);
}
}
```

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

As you can see, the operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.

When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference. Keep in mind that when you create a variable of a class type, you are only creating a reference to an object. Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects are passed to methods by use of call-by-reference. Changes to the object inside the method *do* affect the object used as an argument. For example, consider the following program:

```
// 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
```

Returning Objects

A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is ten greater than it is in the invoking object.

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

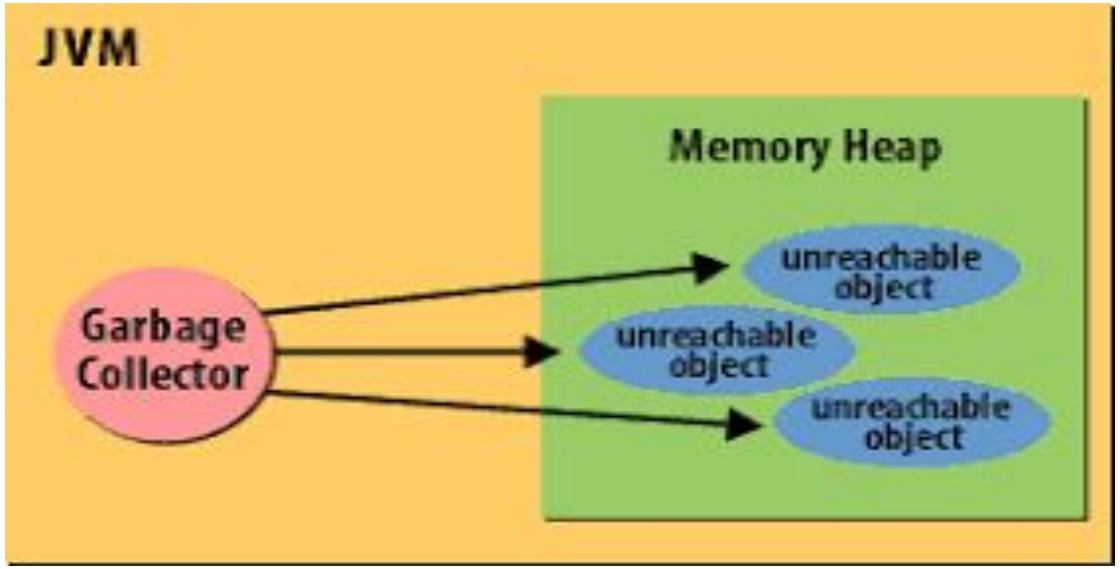
```
class RetOb {
public static void main(String args[]) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incrByTen();
        System.out.println("ob1.a: " + ob1.a);
        System.out.println("ob2.a: " + ob2.a);
        ob2 = ob2.incrByTen();
        System.out.println("ob2.a after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

Garbage Collection

Since objects are dynamically allocated by using the **new** operator, you might be wondering how such objects are destroyed and their memory released for later reallocation. 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.* It works like this: 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 need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used. Furthermore, different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.



```
public class Person {
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public void display() {
    System.out.println("Name: " + name + ", Age: " + age);
  }

  // Overriding finalize() method to perform some actions
before garbage collection
  @Override
  protected void finalize() throws Throwable {
    System.out.println("Object for " + name + " is garbage
collected");
  }

  public static void main(String[] args) {
      Person p1 = new Person("John", 30);
      p1.display();

      // Assigning null to reference variable
      p1 = null;

      // Requesting JVM to run garbage collector
      System.gc();
    }
}
```

In this example, we have defined a class Person with a finalize() method that simply prints a message to the console. In the main() method, we create an object of Person class and call its display() method. Then, we assign null to the reference variable p1 to indicate that the object is no longer needed. Finally, we call the System.gc() method to request the JVM to run the garbage collector.

When the garbage collector runs, it will identify that the Person object created earlier is no longer needed and invoke the finalize() method before reclaiming the memory used by the object. In our example, the finalize() method will print the message "Object for John is garbage collected" to the console.

Example `Scanner` usage

```
import java.util.*;   // so that I can use Scanner

public class ReadSomeInput {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("How old are you? ");
        int age = console.nextInt();

        System.out.println(age + "... That's quite old!");
    }
}
```

- Output (user input underlined):

```
How old are you? 14
14... That's quite old!
```

Another `Scanner` example

```
import java.util.*;   // so that I can use Scanner

public class ScannerSum {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("Please type three numbers: ");
        int num1 = console.nextInt();
        int num2 = console.nextInt();
        int num3 = console.nextInt();

        int sum = num1 + num2 + num3;
        System.out.println("The sum is " + sum);
    }
}
```

- Output (user input underlined):

```
Please type three numbers: 8 6 13
The sum is 27
```

  - The `Scanner` can read multiple values from one line.

Example `Scanner` usage

```
import java.util.*;   // so that I can use Scanner

public class ReadSomeInput {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("whats your name? ");
        String name = console.next();

        System.out.println("your name is "+ name);
    }
}
```

- Output (user input underlined):

```
Whats your name? ram
Your name is ram
```

Example `Scanner` usage

```
import java.util.*;   // so that I can use Scanner

public class ReadSomeInput {
    public static void main(String[] args) {
        Scanner console = new Scanner(System.in);

        System.out.print("whats your college name? ");
        String name = console.nextLine();

        System.out.println("your college name is "+ name);
    }
}
```

- Output (user input underlined):

```
Whats your college name? shri Ramdeobaba college of engg and mgt
Your college name is shri Ramdeobaba college of engg and mgt
```

Module-II (Teaching Hours – 7)

Concept of inheritance, methods of derivation, use of super keyword and final keyword in inheritance, run time polymorphism. Abstract classes and methods, interface, implementation of interface, creating packages, importing packages, static and non-static members.

Understanding static

There will be times when you will want to define a class member that will be used independently of any object of that class. Normally, a class member must be accessed only in conjunction with an object of its class. However, it is possible to create a member that can be used by itself, without reference to a specific instance. To create such a member, precede its declaration with the keyword **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. You can declare both methods and variables to be **static**. 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. When objects of its class are declared, no copy of a **static** variable is made. Instead, all instances of the class share the same **static** variable.

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** or **super** in any way.

If you need to do computation in order to initialize your **static** variables, you can declare a **static** block that gets executed exactly once, when the class is first loaded. The following example shows a class that has a **static** method, some **static** variables, and a **static** initialization block.

```
// 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);
      }
}
```

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, which prints a message and then initializes **b** to **a * 4** or **12**. Then **main( )** is called, which calls **meth( )**, passing **42** to **x**. The three **println( )** statements refer to the two **static** variables **a** and **b**, as well as to the local variable **x**.

Here is the output of the program:

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

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.

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. As you can see, this format is similar to that used to call non-**static** methods through object-reference variables. A **static** variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables.

Here is an example. Inside **main( )**, the **static** method **callme( )** and the **static** variable **b** are accessed through their class name **StaticDemo**.

```
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
```

Inheritance

Inheritance is one of the cornerstones of object-oriented programming because it allows the creation of hierarchical classifications. Using inheritance, you can create a general class that defines traits common to a set of related items. This class can then be inherited by other, more specific classes, each adding those things that are unique to it. In the terminology of Java, a class that is inherited is called a *superclass*. The class that does the inheriting is called a *subclass*. Therefore, a subclass is a specialized version of a superclass. It inherits all of the instance variables and methods defined by the superclass and adds its own, unique elements.

To inherit a class, you simply incorporate the definition of one class into another by using the **extends** keyword.

```
// A simple example of inheritance.
// Create a superclass.
class A
{
	int i, j;
	void showij() {
	System.out.println("i and j: " + i +
	" " + j);
	}
}
// Create a subclass by extending class
A.
class B extends A {
	int k;
	void showk() {
	System.out.println("k: " + k);
	}
	void sum() {
	System.out.println("i+j+k: " +
	(i+j+k));
	}
}
class SimpleInheritance {
public static void main(String
args[]) {
	A superOb = new A();
	B subOb = new B();
	// The superclass may be
	used by itself.
	superOb.i = 10;
	superOb.j = 20;
	System.out.println("Conten
	ts of superOb: ");
	superOb.showij();
	System.out.println();
	/* The subclass has access
	to all public members of
	its superclass. */
	subOb.i = 7;
	subOb.j = 8;
	subOb.k = 9;
	System.out.println("Content
	s of subOb: ");
	subOb.showij();
	subOb.showk();
	System.out.println();
	System.out.println("Sum of
	i, j and k in subOb:");
	subOb.sum();
	}
}
```

The output from this program is shown here:

```
Contents of superOb:
i and j: 10 20
Contents of subOb:
i and j: 7 8
k: 9
Sum of i, j and k in subOb:
i+j+k: 24
```

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. For example, consider the following simple class hierarchy:

```
/* In a class hierarchy, private
members remain
private to their class.
This program contains an error
and will not
compile.
*/
// Create a superclass.
class A {
    int i; // public by default
    private int j; // private to
    A
    void setij(int x, int y) {
    i = x;
    j = y;
    }
}
```

```
// A's j is not accessible here.
class B extends A {
    int total;
    void sum() {
    total = i + j; // ERROR, j is
    not accessible here
    }
}
```

```
class Access {
public static void main(String
args[]) {
    B subOb = new B();
    subOb.setij(10, 12);
    subOb.sum();
    System.out.println("Total is "
    + subOb.total);
    }
}
```

This program will not compile because the reference to **j** inside the **sum( )** method of **B** causes an access violation. Since **j** is declared as **private**, it is only accessible by other members of its own class. Subclasses have no access to it.

```
// This program uses inheritance to extend
Box.
class Box {
double width;
double height;
double depth;
// construct clone of an object
Box(Box ob) { // pass object to constructor
width = ob.width;
height = ob.height;
depth = ob.depth;
}
// constructor used when all dimensions
specified
Box(double w, double h, double d) {
width = w;
height = h;
depth = d;
}
// constructor used when no dimensions
specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is
created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
// Here, Box is extended to include
weight.
class BoxWeight extends Box {
double weight; // weight of box
// constructor for BoxWeight
BoxWeight(double w, double h,
double d, double m) {
width = w;
height = h;
depth = d;
weight = m;
}
}
class DemoBoxWeight {
public static void main(String args[]) {
        BoxWeight mybox1 = new
        BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new
        BoxWeight(2, 3, 4, 0.076);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of
        mybox1 is " + vol);
        System.out.println("Weight of
        mybox1 is " + mybox1.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("Volume of
        mybox2 is " + vol);
        System.out.println("Weight of
        mybox2 is " + mybox2.weight);
        }
}
```

The output from this program is shown here:

```
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3
Volume of mybox2 is 24.0
Weight of mybox2 is 0.076
```

Multilevel Hierarchy

```
// Start with Box.
class Box {
        private double width;
        private double height;
        private double depth;
        // construct clone of an object
        Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
        }
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
        }
        // constructor used when no dimensions specified
        Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
        }
        // constructor used when cube is created
        Box(double len) {
        width = height = depth = len;
        }
        // compute and return volume
        double volume() {
        return width * height * depth;
        }
```

```
// Add weight.
class BoxWeight extends Box {
        double weight; // weight of box
        // construct clone of an object
        BoxWeight(BoxWeight ob) { // pass
        object to constructor
        super(ob);
        weight = ob.weight;
        }
        // constructor when all parameters
        are specified
        BoxWeight(double w, double h,
        double d, double m) {
        super(w, h, d); // call superclass
        constructor
        weight = m;
        }
        // default constructor
        BoxWeight() {
        super();
        weight = -1;
        }
        // constructor used when cube is
        created
        BoxWeight(double len, double m) {
        super(len);
        weight = m;
```

```
// Add shipping costs.
class Shipment extends BoxWeight {
        double cost;
        // construct clone of an object
        Shipment(Shipment ob) { // pass object
        to constructor
        super(ob);
        cost = ob.cost;
        }
        // constructor when all parameters are
        specified
        Shipment(double w, double h, double d,
        double m, double c) {
        super(w, h, d, m); // call superclass
        constructor
        cost = c;
        }
        // default constructor
        Shipment() {
        super();
        cost = -1;
        }
        // constructor used when cube is created
        Shipment(double len, double m, double
        c) {
        super(len, m);
        cost = c;
```

```
class DemoShipment {
public static void main(String args[]) {
        Shipment shipment1 =
        new Shipment(10, 20, 15, 10, 3.41);
        Shipment shipment2 =
        new Shipment(2, 3, 4, 0.76, 1.28);
        double vol;
        vol = shipment1.volume();
        System.out.println("Volume of
        shipment1 is " + vol);
        System.out.println("Weight of
        shipment1 is "
        + shipment1.weight);
        System.out.println("Shipping cost: $" +
        shipment1.cost);
        System.out.println();
        System.out.println("Volume of shipment2 is " + vol);
        System.out.println("Weight of shipment2 is "
        + shipment2.weight);
        System.out.println("Shipping cost: $" +
        shipment2.cost);
        }
}
```

The output of this program is shown here:

```
Volume of shipment1 is 3000.0
Weight of shipment1 is 10.0
Shipping cost: $3.41
Volume of shipment2 is 24.0
Weight of shipment2 is 0.76
Shipping cost: $1.28
```

Using super

In the preceding examples, classes derived from **Box** were not implemented as efficiently or as robustly as they could have been. For example, the constructor for **BoxWeight** explicitly initializes the **width**, **height**, and **depth** fields of **Box( )**. Not only does this duplicate code found in its superclass, which is inefficient, but it implies that a subclass must be granted access to these members. However, there will be times when you will want to create a superclass that keeps the details of its implementation to itself (that is, that keeps its data members private).

In this case, there would be no way for a subclass to directly access or initialize these variables on its own. Since encapsulation is a primary attribute of OOP, it is not surprising that Java provides a solution to this problem. Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**.

super has two general forms. The first calls the superclass' constructor. The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Using super to Call Superclass Constructors

Asubclass can call a constructor defined by its superclass by use of the following form of **super**:
super(*arg-list*);
Here, *arg-list* specifies any arguments needed by the constructor in the superclass. **super( )** must always be the first statement executed inside a subclass' constructor.

To see how **super( )** is used, consider this improved version of the **BoxWeight( )** class:

```
// BoxWeight now uses super to initialize its Box attributes.

class BoxWeight extends Box {
      double weight; // weight of box
      // initialize width, height, and depth using super()
      BoxWeight(double w, double h, double d, double m) {
      super(w, h, d); // call superclass constructor
      weight = m;
      }
}
```

Here, **BoxWeight( )** calls **super( )** with the arguments **w**, **h**, and **d**. This causes the **Box( )** constructor to be called, which initializes **width**, **height**, and **depth** using these values.
BoxWeight no longer initializes these values itself. It only needs to initialize the value unique to it: **weight**. This leaves **Box** free to make these values **private** if desired.

A Second Use for super

The second form of **super** acts somewhat like **this**, except that it always refers to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or an instance variable.

This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass. Consider this simple class hierarchy:

```
// Using super to overcome name hiding.
class A {
        int i;
}
// Create a subclass by extending class A.
class B extends A {
        int i; // this i hides the i in A
        B(int a, int b) {
        super.i = a; // i in A
        i = b; // i in B
}
void show() {
        System.out.println("i in superclass: " + super.i);
        System.out.println("i in subclass: " + i);
        }
}
```

```
class UseSuper {
public static void main(String args[]) {
        B subOb = new B(1, 2);
        subOb.show();
        }
}
```

This program displays the following:

```
i in superclass: 1
i in subclass: 2
```

Although the instance variable **i** in **B** hides the **i** in **A**, **super** allows access to the **i** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

When Constructors Are Called

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called **B** and a superclass called **A**, is **A**'s constructor called before **B**'s, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used. If **super( )** is not used, then the default or parameter less constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A() {
System.out.println("Inside A's constructor.");
}
}
// Create a subclass by extending class A.
class B extends A {
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor.");
}
}
class CallingCons {
public static void main(String args[]) {
        C c = new C();
        }
}
```

The output from this program is shown here:

```
Inside A's constructor
Inside B's constructor
Inside C's constructor
```

Method Overriding

In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to *override* the method in the superclass. When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

```
// Method overriding.
class A {
        int i, j;
        A(int a, int b) {
        i = a;
        j = b;
}
// display i and j
void show() {
        System.out.println("i
        and j: " + i + " " + j);
        }
}
class B extends A {
        int k;
        B(int a, int b, int c) {
        super(a, b);
        k = c;
}
// display k – this overrides show()
in A
void show() {
        System.out.println("k: " + k);
        }
}
class Override {
public static void main(String
args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls
        show() in B
        }
}
```

The output produced by this program is shown here:

k: 3

When **show( )** is invoked on an object of type **B**, the version of **show( )** defined within **B**
is used. That is, the version of **show( )** inside **B** overrides the version declared in **A**.

If you wish to access the superclass version of an overridden method, you can do so by
using **super**.

in this version of **B**, the superclass version of **show( )** is invoked within the subclass' version. This allows all instance variables to be displayed.

```
class B extends A {
        int k;
        B(int a, int b, int c) {
        super(a, b);
        k = c;
        }
        void show() {
        super.show(); // this calls A's
        show()
        System.out.println("k: " + k);
        }
}
```

If you substitute this version of **A** into the previous program, you will see the following output:

```
i and j: 1 2
k: 3
```

Here, **super.show( )** calls the superclass version of **show( )**.

Method overriding occurs *only* when the names and the type signatures of the two methods are identical. If they are not, then the two methods are simply overloaded.

Dynamic Method Dispatch

While the examples in the preceding section demonstrate the mechanics of method overriding, they do not show its power. Indeed, if there were nothing more to method overriding than a name space convention, then it would be, at best, an interesting curiosity, but of little real value. However, this is not the case. Method overriding forms the basis for one of Java's most powerful concepts: *dynamic method dispatch.* Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.

In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
class A {
    void callme() {
    System.out.println("Inside A's
    callme method");
    }
}
class B extends A {
    // override callme()
    void callme() {
    System.out.println("Inside B's
    callme method");
    }
}
class C extends A {
    // override callme()
    void callme() {
    System.out.println("Inside
    C's callme method");
    }
}
class Dispatch {
public static void main(String
args[]) {
    A a = new A(); // object of
    type A
    B b = new B(); // object of
    type B
    C c = new C(); // object of
    type C
    A r; // obtain a reference of type A
    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme
    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme
    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
    }
}
```

The output from the program is shown here:

```
Inside A's callme method
Inside B's callme method
Inside C's callme method
```

This program creates one superclass called **A** and two subclasses of it, called **B** and **C**. Subclasses **B** and **C** override **callme( )** declared in **A**. Inside the **main( )** method, objects of type **A**, **B**, and **C** are declared. Also, a reference of type **A**, called **r**, is declared. The program then in turn assigns a reference to each type of object to **r** and uses that reference to invoke **callme( )**. As the output shows, the version of **callme( )** executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, **r**, you would see three calls to **A**'s **callme( )** method.

Applying Method Overriding

Let's look at a more practical example that uses method overriding. The following program creates a superclass called **Figure** that stores the dimensions of a two-dimensional object. It also defines a method called **area( )** that computes the area of an object. The program derives two subclasses from **Figure**. The first is **Rectangle** and the second is **Triangle**. Each of these subclasses overrides **area( )** so that it returns the area of a rectangle and a triangle, respectively.

```
// Using run-time
polymorphism.
class Figure {
        double dim1;
        double dim2;
        Figure(double a, double
        b) {
        dim1 = a;
        dim2 = b;
        }
        double area() {
        System.out.println("Are
        a for Figure is
        undefined.");
        return 0;
        }
}
```

```
class Rectangle
extends Figure {
        Rectangle(doubl
        e a, double b) {
        super(a, b);
        }
        // override area
        for rectangle
        double area() {
        System.out.printl
        n("Inside Area
        for Rectangle.");
        return dim1 *
        dim2;
        }
}
```

```
class Triangle extends
Figure {
        Triangle(double a,
        double b) {
        super(a, b);
        }
        // override area
        for right triangle
        double area() {
        System.out.println
        ("Inside Area for
        Triangle.");
        return dim1 *
        dim2 / 2;
        }
}
```

```
class FindAreas {
public static void main(String
args[]) {
Figure f = new Figure(10, 10);
Rectangle r = new Rectangle(9,
5);
Triangle t = new Triangle(10, 8);
Figure figref;
figref = r;
System.out.println("Area is " +
figref.area());
figref = t;
System.out.println("Area is " +
figref.area());
figref = f;
System.out.println("Area is " +
figref.area());
}
}
```

The output from the program is shown here:

```
Inside Area for Rectangle.
Area is 45
Inside Area for Triangle.
Area is 40
Area for Figure is undefined.
Area is 0
```

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects.

In this case, if an object is derived from **Figure**, then its area can be obtained by calling **area( )**.
The interface to this operation is the same no matter what type of figure is being used.

Using Abstract Classes

There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method. That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details. Such a class determines the nature of the methods that the subclasses must implement. One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method. This is the case with the class **Figure** used in the preceding example. The definition of **area( )** is simply a placeholder. It will not compute and display the area of any type of object.

As you will see as you create your own class libraries, it is not uncommon for a method to have no meaningful definition in the context of its superclass. You can handle this situation two ways. One way, as shown in the previous example, is to simply have it report a warning message. While this approach can be useful in certain situations—such as debugging—it is not usually appropriate. You may have methods that must be overridden by the subclass in order for the subclass to have any meaning. Consider the class **Triangle**. It has no meaning if **area( )** is not defined. In this case, you want some way to ensure that a subclass does, indeed, override all necessary methods. Java's solution to this problem is the *abstract method.*

You can require that certain methods be overridden by subclasses by specifying the **abstract** type modifier. These Methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the superclass. Thus, a subclass must override them—it cannot simply use the version defined in the superclass. To declare an abstract method, use this general form:

abstract *type name(parameter-list)*;

As you can see, no method body is present.

Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, you simply use the **abstract** keyword in front of the **class** keyword at the beginning of the class declaration. There can be no objects of an abstract class. That is, an abstract class cannot be directly instantiated with the **new** operator. Such objects would be useless, because an abstract class is not fully defined. Also, you cannot declare abstract constructors, or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared **abstract**.

Here is a simple example of a class with an abstract method, followed by a class which implements that method:

```
// A Simple demonstration of abstract.
abstract class A {
	abstract void callme();
	// concrete methods are still allowed in abstract classes
	void callmetoo() {
	System.out.println("This is a concrete method.");
	}
}
class B extends A {
	void callme() {
	System.out.println("B's implementation of callme.");
	}
}
```

```
class AbstractDemo {
public static void main(String args[]) {
	B b = new B();
	b.callme();
	b.callmetoo();
	}
}
```

Notice that no objects of class **A** are declared in the program. As mentioned, it is not possible to instantiate an abstract class. One other point: class **A** implements a concrete method called **callmetoo( )**. This is perfectly acceptable. Abstract classes can include as much implementation as they see fit. Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of superclass references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Using final with Inheritance

The keyword **final** has three uses. First, it can be used to create the equivalent of a named constant. This use was described in the preceding chapter. The other two uses of **final** apply to inheritance. Both are examined here.

Using final to Prevent Overriding

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden. The following fragment illustrates **final**:

```
class A {
    final void meth() {
    System.out.println("This is a final method.");
    }
}
class B extends A {
    void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
    }
}
```

Because **meth( )** is declared as **final**, it cannot be overridden in **B**. If you attempt to do so, a compile-time error will result.

Using final to Prevent Inheritance

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with **final**. Declaring a class as **final** implicitly declares all of its methods as **final**, too. As you might expect, it is illegal to declare a class as both **abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide
complete implementations.

Here is an example of a **final** class:

```
final class A {
// ...
}
// The following class is illegal.
class B extends A { // ERROR! Can't subclass A
// ...
}
```

As the comments imply, it is illegal for **B** to inherit **A** since **A** is declared as **final**.

Interfaces

Using the keyword **interface**, you can fully abstract a class' interface from its implementation. That is, using **interface**, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body. In practice, this means that you can define interfaces that don't make assumptions about how they are implemented. Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism

Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible. This requirement by itself makes for a static and non extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses.

Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared. When it is declared as **public**, the interface can be used by any other code. In this case, the interface must be the only public interface declared in the file, and the file must have the same name as the interface. *name* is the name of the interface, and can be any valid identifier. Notice that the methods that are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized. All methods and variables are implicitly **public**.

```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass]
[implements interface [,interface...]] {
// class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

Here is a small example class that implements the **Callback** interface shown earlier.

```
class Client implements Callback {
// Implement Callback's interface
      public void callback(int p) {
      System.out.println("callback called with " + p);
      }
}
```

Notice that **callback( )** is declared using the **public** access specifier.

Interface example

```
interface Bank{
	float rateOfInterest();
}
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
	SBI b=new SBI();
	System.out.println("ROI: "+b.rateOfInterest());

	PNB b1=new PNB();
	System.out.println("ROI: "+b.rateOfInterest());
	}
}
```

Output:

9.15

Variables in Interfaces

```
interface Try
{
   int a=10;
   public int a=10;
   public static final int a=10;
   final int a=10;
   static int a=0;
}
```

All of the above statements are identical

```
class Sample implements Try
{
  public static void main(String args[])
  {
     a=20; //compile time error
  }
}
```

Inside any implementation class, you cannot change the variables declared in interface because by default, they are public, static and final. Here we are implementing the interface "Try" which has a variable a. When we tried to set the value for variable a we got compilation error as the variable a is public static **final** by default and final variables can not be re-initialized.

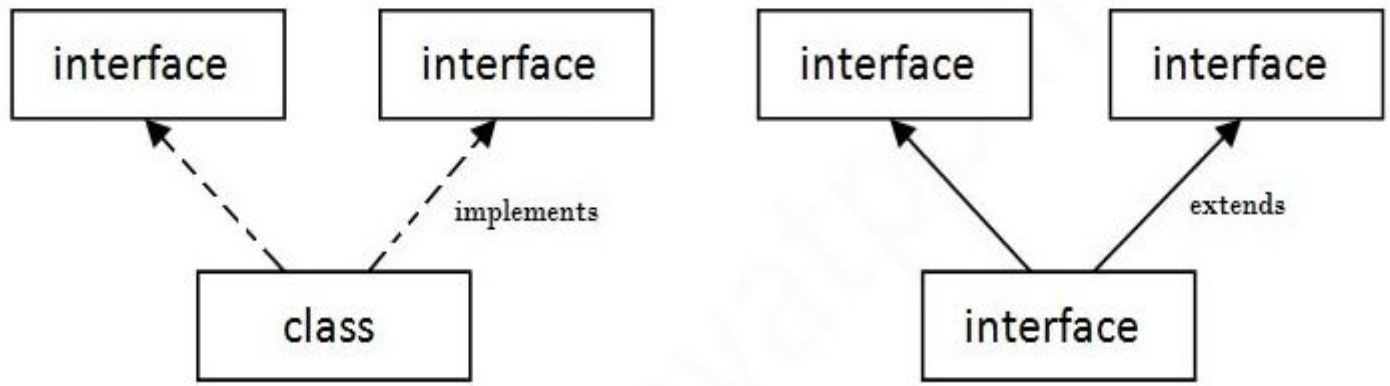
An interface can extend any interface but cannot implement it. Class implements interface and interface extends interface.

A **class** can implement any **number of interfaces**.

If there are **two or more same methods** in two interfaces and a class implements both interfaces, implementation of the method once is enough.

Variable names conflicts can be resolved by interface name

```
interface A
{
   int x=10;
}
interface B
{
   int x=100;
}
class Hello implements A,B
{
   public static void main(String args[])
   {
     /* reference to x is ambiguous both variables are x
      * so we are using interface name to resolve the
      * variable
      */
     //System.out.println(x);
     System.out.println(A.x);
     System.out.println(B.x);
   }
}
```



Multiple Inheritance in Java

Multiple inheritance in Java by interface

```
interface Printable
{
	void print();
}
interface Showable
{
	void show();
}
class A7 implements Printable,Showable
{
	public void print()
	{
		System.out.println("Hello");
	}
	public void show()
	{
		System.out.println("Welcome");
	}
public static void main(String args[])
{
	A7 obj = new A7();
	obj.print();
	obj.show();
 }
}
```

Output:

```
Hello
Welcome
```

A class implements interface but one interface extends another interface .

```
 interface Printable
{
    void print();
 }
 interface Showable extends Printable
{
    void show();
 }

class TestInterface4 implements Showable
{
 public void print()
 {
    System.out.println("Hello");
 }
 public void show()
 {
    System.out.println("Welcome");
  }

public static void main(String args[])
{
 TestInterface4 obj = new TestInterface4();
 obj.print();
 obj.show();
  }
 }
```

Output:

Hello

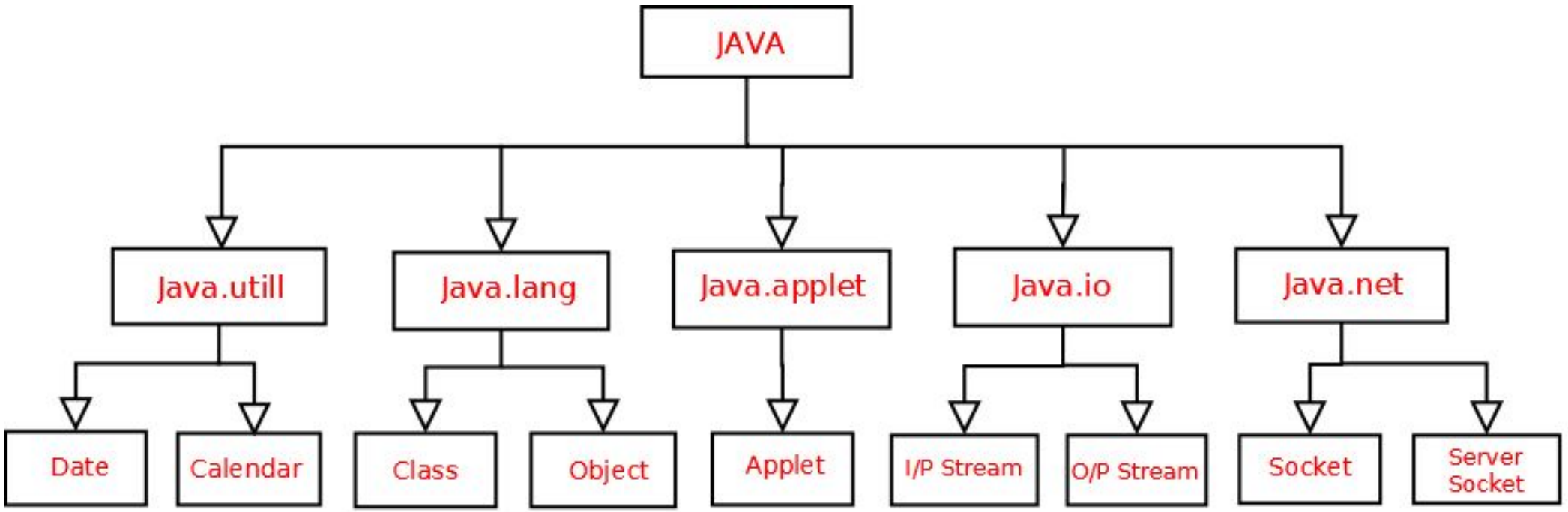
Welcome

Interfaces Can Be Extended

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
        void meth1();
        void meth2();
}
// B now includes meth1() and meth2() -- it
adds meth3().
interface B extends A {
        void meth3();
}
// This class must implement all of A and B
class MyClass implements B {
public void meth1() {
System.out.println("Implement meth1().");
}
public void meth2()
 {
        System.out.println("Implement meth2().");
}
public void meth3()
{
        System.out.println("Implement meth3().");
}
}
class IFExtend {
        public static void main(String arg[]) {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

Packages


JAVA
Java.utill
Java.lang
Java.applet
Java.io
Java.net
Date
Calendar
Class
Object
Applet
I/P Stream
O/P Stream
Socket
Server Socket

Packages are containers for classes that are used to keep the class name space compartmentalized. For example, a package allows you to create a class named List, which you can store in your own package without concern that it will collide with some other class named List stored elsewhere.

The name of each example class was taken from the same name space. This means that a unique name had to be used for each class to avoid name collisions. After a while, without some way to manage the name space, you could run out of convenient, descriptive names for individual classes. You also need some way to be assured that the name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers. (Imagine a small group of programmers fighting over who gets to use the name “Foobar” as a class name. Or, imagine the entire Internet community arguing over who first named a class “Espresso.”) Thankfully, Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package. The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

Defining a Package

To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will be long to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code. This is the general form of the package statement:

```
package pkg;
```

Here, pkg is the name of the package. For example, the following statement creates a package called MyPackage.

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name fromtheoneaboveitbyuseofaperiod.Thegeneralformofamultileveledpackagestatement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

A Short Package Example Keeping the preceding discussion in mind, you can try this simple package:

```
// A simple package
package MyPack;
class Balance {
	String name;
	double bal;
	Balance(String n, double b)
	{
		name = n;
		bal = b;
	}
	void show() {
		if(bal<0)
		System.out.print("--> ");
		System.out.println(name + ": $" + bal);
	}
}
class AccountBalance{
	public static void main(String args[]) {
		Balance current[] = new Balance[3];
		current[0] = new Balance("K. J. Fielding", 123.23);
		current[1] = new Balance("Will Tell", 157.02);
		current[2] = new Balance("Tom Jackson", -12.33);
		for(int i=0; i<3; i++)
		current[i].show();
		}
}
```

Call this file AccountBalance.java and put it in a directory called MyPack. Next, compile the file. Make sure that the resulting .class file is also in the MyPack directory. Then, try executing the AccountBalance class, using the following command line:

```
java MyPack.AccountBalance.
```

As explained, AccountBalance is now part of the package MyPack. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

Packages add another dimension to access control. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. Following table sums up the interactions. While Java's access control mechanism may seem complicated, we can simplify it as follows.

Anything declared public can be accessed from anywhere.
Anything declared private cannot be seen outside of its class.
When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access.
If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element protected.

When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

An Access Example

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages—in this case, **p1** and **p2**.

The source for the first package defines three classes: **Protection**, **Derived**, and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is **private**, **n_pro** is **protected**, and **n_pub** is **public**.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants **Derived** access to every variable in **Protection** except for **n_pri**, the **private** one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

This is file Protection.java

```
package p1;
public class Protection
{
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;

public Protection()
{       System.out.println("base constructor");
System.out.println("n = " + n);
System.out.println("n_pri = " + n_pri);
System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub); }
}
```

This is file Derived.java

```
package p1;
class Derived extends Protection {
Derived()
{
        System.out.println("derived constructor");
System.out.println("n = " + n);
//  class only //  System.out.println("n_pri = "4 + n_pri);
        System.out.println("n_pro = " + n_pro);
System.out.println("n_pub = " + n_pub);
        }
}
```

This is file SamePackage.java

```
package p1;
class SamePackage
{
	SamePackage()
	{
		Protection p = new Protection(); System.out.println("same package constructor");
		System.out.println("n = " + p.n);
		//  class only //  System.out.println("n_pri = " + p.n_pri);
		System.out.println("n_pro = " + p.n_pro); System.out.println("n_pub = " + p.n_pub);
	}
}
```

Following is the source code for the other package, p2. The two classes defined in p2 cover the other two conditions that are affected by access control. The first class, Protection2, is a subclass of p1.Protection. This grants access to all of p1.Protection's variables except for n_pri(because it is private) and n, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package subclasses. Finally, the class OtherPackage has access to only one variable, n_pub, which was declared public.

This is file Protection2.java:

```
package p2;
class Protection2 extends p1.Protection
 {
       Protection2()
       {
            System.out.println("derived other package
           constructor");
           //  class or package only
           //  System.out.println("n = " + n);
           //  class only
           //  System.out.println("n_pri = " + n_pri);
           System.out.println("n_pro = " + n_pro);
           System.out.println("n_pub = " + n_pub);
      }
}
```

This is file OtherPackage.java:

```
package p2;
class OtherPackage
 {
       OtherPackage()
        {
            p1.Protection p = new p1.Protection();
            System.out.println("other package constructor");
            //  class or package only
            //  System.out.println("n = " + p.n);
            //  class only
            //  System.out.println("n_pri = " + p.n_pri);
            //  class, subclass or package only
             //  System.out.println("n_pro = " + p.n_pro);
            System.out.println("n_pub = " + p.n_pub);
       }
}
```

If you wish to try these two packages, here are two test files you can use. The one for package p1 is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
 {
      public static void main(String args[])
       {
            Protection ob1 = new Protection();
            Derived ob2 = new Derived();
            SamePackage ob3 = new SamePackage();
       }
 }
```

The test file for p2 is shown next:

```
// Demo package p2.
package p2;
// Instantiate the various classes in p2.
public class Demo
 {
      public static void main(String args[])
      {
            Protection2 ob1 = new Protection2();
           OtherPackage ob2 = new OtherPackage();
      }
}
```

Importing the packages

In a Java source file,import statements occur immediately following the package statement (if it exists) and before any class definitions. This Is the general form of the import statement:
import pkg1[.pkg2].(classname|*);

```
import java.util.Date;
 import java.io.*;
```

```
import java.util.*;
class MyDate extends Date {     }
```

The same example without the import statement looks like this:

```
class MyDate extends java.util.Date {    }
```

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in java.lang, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code. For example, if you want the Balance class of the package MyPack shown earlier to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

```
package MyPack;
/* Now, the Balance class, its constructor, and its show() method
are public.  This means that they can be used by non-subclass
code outside their package. */
public class Balance {
String name;
double bal;
public Balance(String n, double b)
{
        name = n;
        bal = b;
        }
        public void show()
        {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
} }
```

```
import MyPack.*;
class TestBalance
 {
public static void main(String args[]) {
/* Because Balance is public, you may use Balance
class and call its constructor. */
Balance test = new Balance("J. J. Jaspers", 99.88);
test.show(); // you may also call show()
}
}
```

As an experiment, remove the public specifier from the Balance class and then try compiling TestBalance. As explained, errors will result.

Module-III (Teaching Hours - 8)

Exceptions, types of exception, use of try catch block, handling multiple exceptions, using finally, throw and throws clause, user defined exceptions, Generics, generic class with two type parameter, bounded generics, Collection classes: Arrays, Vectors, Array list, Linked list, Hash set, Queues, Trees.

Exception Handling

An *exception* is an abnormal condition that arises in a code sequence at run time.

In other words, an exception is a run-time error.

In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.

When an exceptional condition arises, an object representing that exception is created and *thrown* in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is *caught* and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work.

Program statements that you want to monitor for exceptions are contained within a **try** block.

If an exception occurs within the **try** block, it is thrown. Your code can catch this exception (using **catch**) and handle it in some rational manner.

System-generated exceptions are automatically thrown by the Java run-time system.

To manually throw an exception, use the keyword **throw**.

Any exception that is thrown out of a method must be specified as such by a **throws** clause.

Any code that absolutely must be executed after a **try** block completes is put in a **finally** block.

This is the general form of an exception-handling block:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

Here, *ExceptionType* is the type of exception that has occurred.

Exception Types

All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy. Immediately below **Throwable** are two subclasses that partition exceptions into two distinct branches. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by **Error**, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

Uncaught Exceptions

Before you learn how to handle exceptions in your program, it is useful to see what happens when you don't handle them. This small program includes an expression that intentionally causes a divide-by-zero error:

```
class Exc0 {
public static void main(String args[]) {
      int d = 0;
      int a = 42 / d;
      }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of **Exc0** to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. In this example, we haven't supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system. Any exception that is not caught by your program will ultimately be processed by the default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program.

Here is the exception generated when this example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:4)
```

Notice how the class name, **Exc0**; the method name, **main**; the filename, **Exc0.java**; and the line number, **4**, are all included in the simple stack trace.

Also, notice that the type of exception thrown is a subclass of **Exception** called **ArithmeticException**, which more specifically describes what type of error happened. As discussed later in this chapter, Java supplies several built-in exception types that match the various sorts of run-time errors that can be generated.

The stack trace will always show the sequence of method invocations that led up to the error.

For example, here is another version of the preceding program that introduces the same error but in a method separate from **main( )**:

```
class Exc1 {
static void subroutine() {
        int d = 0;
        int a = 10 / d;
}
public static void main(String args[]) {
        Exc1.subroutine();
        }
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:4)
at Exc1.main(Exc1.java:7)
```

As you can see, the bottom of the stack is **main**'s line 7, which is the call to **subroutine( )**, which caused the exception at line 4. The call stack is quite useful for debugging, because it pinpoints the precise sequence of steps that led to the error.

Using try and catch

The following program includes a **try** block and a **catch** clause that processes the **ArithmeticException** generated by the division-by-zero error:

```
class Exc2 {
	public static void main(String args[]) {
		int d, a;
		try
		 { // monitor a block of code.
			d = 0;
			a = 42 / d;
			System.out.println("This will not be printed.");
			}
			catch (ArithmeticException e)
			 { // catch divide-by-zero error
				System.out.println("Division by zero.");
		}
		System.out.println("After catch statement.");
	}
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to **println( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try**/**catch** mechanism.

A **try** and its **catch** statement form a unit. The scope of the **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.

The goal of most well-constructed **catch** clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the **for** loop obtains two random integers. Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into **a**. If either division operation causes a divide-by-zero error, it is caught, the value of **a** is set to zero, and the program continues.

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
	public static void main(String args[])
	{
		int a=0, b=0, c=0;
		Random r = new Random();
		for(int i=0; i<32000; i++) {
		try {
		b = r.nextInt();
		c = r.nextInt();
		a = 12345 / (b/c);
		}
		catch (ArithmeticException e) {
		System.out.println("Division by zero.");
		a = 0; // set a to zero and continue
		}
		System.out.println("a: " + a);
		}
	}
}
```

Displaying a Description of an Exception

Throwable overrides the **toString( )** method (defined by **Object**) so that it returns a string containing a description of the exception. You can display this description in a **println( )** statement by simply passing the exception as an argument. For example, the **catch** block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e) {
      System.out.println("Exception: " + e);
      a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by zero error displays the following message:

```
Exception: java.lang.ArithmeticException: / by zero
```

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try**/**catch**
block.

```
// Demonstrate multiple catch statements.
class MultiCatch {
        public static void main(String args[]) {
        try {
                int a = args.length;
                System.out.println("a = " + a);
                int b = 42 / a;
                int c[] = { 1 };
                c[42] = 99;
        }
        catch(ArithmeticException e) {
                System.out.println("Divide by 0: " + e);
        }
        catch(ArrayIndexOutOfBoundsException e) {
                System.out.println("Array index oob: " + e);
        }
        System.out.println("After try/catch blocks.");
        }
}
```

This program will cause a division-by-zero exception if it is started with no commandline arguments, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the program attempts to assign a value to **c[42]**.

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

When you use multiple **catch** statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```
/* This program contains an error.
A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.
*/
class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception.
*/
catch(ArithmeticException e) { // ERROR -
unreachable
System.out.println("This is never reached.");
}
}
}
```

Throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for your program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

throw *ThrowableInstance*;

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**.

Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
There are two ways you can obtain a **Throwable** object:

using a parameter in a **catch** clause, or creating one with the **new** operator. The flow of execution stops immediately after the **throw** statement; any subsequent statements are not executed. The nearest enclosing **try** block is inspected to see if it has a **catch** statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing **try** statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program and prints the stack trace.

Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

```
// Demonstrate throw.
class ThrowDemo {
	static void demoproc() {
		try {
			throw new NullPointerException("demo");
		} catch(NullPointerException e) {
			System.out.println("Caught inside demoproc.");
			throw e; // rethrow the exception
			}
		}
		public static void main(String args[]) {
		try {
			demoproc();
		} catch(NullPointerException e) {
			System.out.println("Recaught: " + e);
		}
	}
}
```

This program gets two chances to deal with the same error. First, **main( )** sets up an exception context and then calls **demoproc( )**. The **demoproc( )** method then sets up another exception handling context and immediately throws a new instance of **NullPointerException**, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects. Pay close attention to this line:

```
throw new NullPointerException("demo");
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's builtin run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print( )** or **println( )**.

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

This is the general form of a method declaration that includes a **throws** clause:

type method-name(*parameter-list*) throws *exception-list*
{
// body of method
}
Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.

class ThrowsDemo {
	static void throwOne() {
		System.out.println("Inside throwOne.");
		throw new IllegalAccessException("demo");
	}
	public static void main(String args[]) {
		throwOne();
	}
}
```

To make this example compile, you need to make two changes. First, you need to declare that **throwOne( )** throws **IllegalAccessException**. Second, **main( )** must define a **try**/**catch** statement that catches this exception.

```
// This is now correct.

class ThrowsDemo {
      static void throwOne() throws IllegalAccessException {
            System.out.println("Inside throwOne.");
            throw new IllegalAccessException("demo");
      }
      public static void main(String args[]) {
            try {
                  throwOne();
      } catch (IllegalAccessException e) {
            System.out.println("Caught " + e);
      }
   }
}
```

Here is the output generated by running this example program:

```
inside throwOne
caught java.lang.IllegalAccessException: demo
```

Finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.

For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be by passed by the exception-handling mechanism. The **finally** keyword is designed to address this contingency. **finally** creates a block of code that will be executed after a **try**/**catch** block has completed and before the code following the **try/catch** block.

The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns.

This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

Here is an example program that shows three methods that exit in various ways, none without executing their **finally** clauses:

```
// Demonstrate finally.
class FinallyDemo {
// Through an exception out of the method.
static void procA() {
try {
       System.out.println("inside procA");
       throw new RuntimeException("demo");
     }
finally {
       System.out.println("procA's finally");
          }
}
// Return from within a try block.
static void procB() {
try {
       System.out.println("inside procB");
       return;
      }
finally {
       System.out.println("procB's finally");
          }
}
// Execute a try block normally.
static void procC() {
try {
       System.out.println("inside procC");
} finally {
       System.out.println("procC's finally");
}}
public static void main(String args[]) {
try {
       procA();
} catch (Exception e) {
       System.out.println("Exception caught");
}
procB();
procC();
}}
```

In this example, **procA( )** prematurely breaks out of the **try** by throwing an exception. The **finally** clause is executed on the way out. **procB( )**'s **try** statement is exited via a **return** statement. The **finally** clause is executed before **procB( )** returns. In **procC( )**, the **try** statement executes normally, without error. However, the **finally** block is still executed.

REMEMBER *If a **finally** block is associated with a **try**, the **finally** block will be executed upon conclusion of the **try**.*

Here is the output generated by the preceding program:

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions are subclasses of the standard type **RuntimeException**. As previously explained, these exceptions need not be included in any method's **throws** list.

In the language of Java, these are called *unchecked exceptions* because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in **java.lang** are listed in Table 10-1.

Table 10-2 lists those exceptions defined by **java.lang** that must be included in a method's **throws** list if that method can generate one of these exceptions and does not handle it itself. These are called *checked exceptions*. Java defines several other types of exceptions that relate to its various class libraries.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

TABLE 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

TABLE 10-2 Java's Checked Exceptions Defined in **java.lang**

Creating Your Own Exception Subclasses

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).

Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The **Exception** class does not define any methods of its own. It does, of course, inherit those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them. They are shown in Table 10-3.

Method	Description
Throwable fillInStackTrace( )	Returns a **Throwable** object that contains a completed stack trace. This object can be rethrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage( )	Returns a description of the exception.
StackTraceElement[ ] getStackTrace( )	Returns an array that contains the stack trace, one element at a time, as an array of **StackTraceElement**. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The **StackTraceElement** class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable *causeExc*)	Associates *causeExc* with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.
void printStackTrace(PrintStream *stream*)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter *stream*)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement *elements*[ ])	Sets the stack trace to the elements passed in *elements*. This method is for specialized applications, not normal use.
String toString( )	Returns a **String** object containing a description of the exception. This method is called by **println( )** when outputting a **Throwable** object.

TABLE 10-3 The Methods Defined by **Throwable**

You may also wish to override one or more of these methods in exception classes that you create.

The following example declares a new subclass of **Exception** and then uses that subclass to signal an error condition in a method. It overrides the **toString( )** method, allowing a carefully tailored description of the exception to be displayed.

```
// This program creates a custom exception type.

class MyException extends Exception {
	private int detail;
	MyException(int a) {
		detail = a;
		}
	public String toString() {
		return "MyException[" + detail + "]";
	}
}
```

```
class ExceptionDemo
{
	static void compute(int a) throws MyException
	{
		System.out.println("Called compute(" + a + ")");
		if(a > 10)
		throw new MyException(a);
		System.out.println("Normal exit");
	}
	public static void main(String args[])
	{
		try {
			compute(1);
			compute(20);
		       }
		catch (MyException e)
		{
			System.out.println("Caught " + e);
		}
	}
}
```

This example defines a subclass of **Exception** called **MyException**. This subclass is quite simple: it has only a constructor plus an overloaded **toString( )** method that displays the value of the exception. The **ExceptionDemo** class defines a method named **compute( )** that throws a **MyException** object. The exception is thrown when **compute( )**'s integer parameter is greater than 10. The **main( )** method sets up an exception handler for **MyException**, then calls **compute( )** with a legal value (less than 10) and an illegal one to show both paths through the code.

Here is the result:

```
Called compute(1)
Normal exit
Called compute(20)
Caught MyException[20]
```

Generics, generic class with two type parameter, bounded generics

Fortunately, generics are not difficult to use, and they provide significant benefits for the Java programmer. Through the use of generics, it is possible to create classes, interfaces, and methods that will work in a type-safe manner with various kinds of data. Many algorithms are logically the same no matter what type of data they are being applied to. For example, the mechanism that supports a stack is the same whether that stack is storing items of type **Integer**, **String**, **Object**, or **Thread**.

With generics, you can define an algorithm once, independently of any specific type of data, and then apply that algorithm to a wide variety of data types without any additional effort. The expressive power generics add to the language fundamentally changes the way that Java code is written.

What Are Generics?

At its core, the term *generics* means *parameterized types.* Parameterized types are important because they enable you to create classes, interfaces, and methods in which the type of data upon which they operate is specified as a parameter. Using generics, it is possible to create a single class, for example, that automatically works with different types of data.

A class, interface, or method that operates on a parameterized type is called *generic,* as in *generic class* or *generic method.*

A Simple Generics Example

Let's begin with a simple example of a generic class. The following program defines two classes. The first is the generic class **Gen**, and the second is **GenDemo**, which uses **Gen**.

```
// A simple generic class.
// Here, T is a type parameter that
// will be replaced by a real type
// when an object of type Gen is created.
class Gen<T> {
T ob; // declare an object of type T
// Pass the constructor a reference to
// an object of type T.
Gen(T o) {
ob = o;
}
// Return ob.
T getob() {
return ob;
}
// Show type of T.
void showType() {
System.out.println("Type of T is " +
ob.getClass().getName());
}
}
```

```
// Demonstrate the generic class.
class GenDemo {
public static void main(String args[]) {
// Create a Gen reference for Integers.
Gen<Integer> iOb;
// Create a Gen<Integer> object and assign its reference to iOb. //Notice
the use of autoboxing to encapsulate the value 88 //within an Integer
object.
iOb = new Gen<Integer>(88);
// Show the type of data used by iOb.
iOb.showType();
// Get the value in iOb. Notice that no cast is needed.
int v = iOb.getob();
System.out.println("value: " + v);
System.out.println();
// Create a Gen object for Strings.
Gen<String> strOb = new Gen<String>("Generics Test");
// Show the type of data used by strOb.
strOb.showType();
// Get the value of strOb. Again, notice that no cast is needed.
String str = strOb.getob();
System.out.println("value: " + str);
}
}
```

The output produced by the program is shown here:

```
Type of T is java.lang.Integer
value: 88
Type of T is java.lang.String
value: Generics Test
```

Let's examine this program carefully.
First, notice how **Gen** is declared by the following line:
class Gen<T> {
Here, **T** is the name of a *type parameter.*

Next, **T** is used to declare an object called **ob**, as shown here:

```
T ob; // declare an object of type T
```

Now consider **Gen**'s constructor:

```
Gen(T o) {
ob = o;
}
```

Etc…

Generics Work Only with Objects

When declaring an instance of a generic type, the type argument passed to the type parameter must be a class type. You cannot use a primitive type, such as **int** or **char**. For example, with **Gen**, it is possible to pass any class type to **T**, but you cannot pass a primitive type to a type parameter.

Therefore, the following declaration is illegal:

```
Gen<int> strOb = new Gen<int>(53); // Error, can't use primitive type
```

Generic Types Differ Based on Their Type Arguments

A key point to understand about generic types is that a reference of one specific version of a generic type is not type compatible with another version of the same generic type. For example, assuming the program just shown, the following line of code is in error and will not compile:

```
iOb = strOb; // Wrong!
```

A Generic Class with Two Type Parameters

You can declare more than one type parameter in a generic type. To specify two or more type parameters, simply use a comma-separated list. For example, the following **TwoGen** class is a variation of the **Gen** class that has two type parameters:

```
// A simple generic class with two type
// parameters: T and V.
class TwoGen<T, V> {
T ob1;
V ob2;
// Pass the constructor a reference to
// an object of type T and an object of type V.
TwoGen(T o1, V o2) {
ob1 = o1;
ob2 = o2;
}
// Show types of T and V.
void showTypes() {
System.out.println("Type of T is " +
ob1.getClass().getName());
System.out.println("Type of V is " +
ob2.getClass().getName());
}
T getob1() {
return ob1;
}
V getob2() {
return ob2;
}
}
```

```
// Demonstrate TwoGen.
class SimpGen {
public static void main(String args[]) {
TwoGen<Integer, String> tgObj =
new TwoGen<Integer, String>(88, "Generics");
// Show the types.
tgObj.showTypes();
// Obtain and show values.
int v = tgObj.getob1();
System.out.println("value: " + v);
String str = tgObj.getob2();
System.out.println("value: " + str);
}
}
```

The output from this program is shown here:

```
Type of T is java.lang.Integer
Type of V is java.lang.String
value: 88
value: Generics
```

In this case, **Integer** is substituted for **T**, and **String** is substituted for **V**.

The General Form of a Generic Class

The generics syntax shown in the preceding examples can be generalized. Here is the syntax for declaring a generic class:

class *class-name*<*type-param-list*> { // ...

Here is the syntax for declaring a reference to a generic class:

class-name<*type-arg-list*> *var-name* = new *class-name*<*type-arg-list*>(*cons-arg-list*);

Creating a Generic Method

As the preceding examples have shown, methods inside a generic class can make use of a class' type parameter and are, therefore, automatically generic relative to the type parameter.

However, it is possible to declare a generic method that uses one or more type parameters of its own. Furthermore, it is possible to create a generic method that is enclosed within a non-generic class.

Let's begin with an example. The following program declares a non-generic class called **GenMethDemo** and a static generic method within that class called **isIn( )**. The **isIn( )** method determines if an object is a member of an array. It can be used with any type of object and array as long as the array contains objects that are compatible with the type of the object being sought.

```
// Demonstrate a simple generic method.
class GenMethDemo {
    // Determine if an object is in an array.
    static <T, V extends T> boolean isIn(T x, V[] y)
    {
        for(int i=0; i < y.length; i++)
        if(x.equals(y[i]))
            return true;

        return false;
    }
    public static void main(String args[]) {
        // Use isIn() on Integers.
        Integer nums[] = { 1, 2, 3, 4, 5 };

        if(isIn(2, nums))
        System.out.println("2 is in nums");

        if(!isIn(7, nums))
        System.out.println("7 is not in nums");
        System.out.println();

        // Use isIn() on Strings.
        String strs[] = { "one", "two", "three","four", "five" };

        if(isIn("two", strs))
        System.out.println("two is in strs");

        if(!isIn("seven", strs))
        System.out.println("seven is not in strs");

        // Oops! Won't compile! Types must be compatible.
        // if(isIn("two", nums))
        // System.out.println("two is in strs");
    }
}
```

The output from the program is shown here:

```
2 is in nums
7 is not in nums
two is in strs
seven is not in strs
```

```
//: generics/GenericMethods.java
public class GenericMethods {
public <T> void f(T x) {
System.out.println(x.getClass().getName());
}
public static void main(String[] args) {
GenericMethods gm = new GenericMethods();
gm.f("");
gm.f(1);
gm.f(1.0);
gm.f(1.0F);
gm.f('c');
gm.f(gm);
}
} /* Output:
java.lang.String
java.lang.Integer
java.lang.Double
java.lang.Float
java.lang.Character
GenericMethods
```

Let's examine **isIn( )** closely. First, notice how it is declared by this line:

static <T, V extends T> boolean isIn(T x, V[] y) {

The type parameters are declared *before* the return type of the method.

Second, notice that the type **V** is upper-bounded by **T**. Thus, **V** must either be the same as type **T**, or a subclass of **T**. This relationship enforces that **isIn( )** can be called only with arguments that are compatible with each other. Also notice that **isIn( )** is static, enabling it to be called independently of any object. Understand, though, that generic methods can be either static or non-static. There is no restriction in this regard.

The syntax used to create **isIn( )** can be generalized. Here is the syntax for a generic method:

<*type-param-list*> *ret-type meth-name*(*param-list*) { // ...

In all cases, *type-param-list* is a comma-separated list of type parameters. Notice that for a generic method, the type parameter list precedes the return type.

Bounded Types

In the preceding examples, the type parameters could be replaced by any class type. This is fine for many purposes, but sometimes it is useful to limit the types that can be passed to a type parameter. For example, assume that you want to create a generic class that contains a method that returns the average of an array of numbers. Furthermore, you want to use the class to obtain the average of an array of any type of number, including integers, floats, and doubles. Thus, you want to specify the type of the numbers generically, using a type parameter.

```
// In this version of Stats, the type argument for
// T must be either Number, or a class derived
// from Number.
class Stats<T extends Number> {
T[] nums; // array of Number or subclass
// Pass the constructor a reference to
// an array of type Number or subclass.
Stats(T[] o) {
nums = o;
}
```

```
// Return type double in all cases.
double average() {
double sum = 0.0;
for(int i=0; i < nums.length; i++)
sum += nums[i].doubleValue();
return sum / nums.length;
}
}
```

```
// Demonstrate Stats.
class BoundsDemo {
public static void main(String args[]) {

Integer inums[] = { 1, 2, 3, 4, 5 };
Stats<Integer> iob = new Stats<Integer>(inums);
double v = iob.average();
System.out.println("iob average is " + v);

Double dnums[] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
Stats<Double> dob = new Stats<Double>(dnums);
double w = dob.average();
System.out.println("dob average is " + w);

// This won't compile because String is not a
// subclass of Number.
// String strs[] = { "1", "2", "3", "4", "5" };
// Stats<String> strob = new Stats<String>(strs);
// double x = strob.average();
// System.out.println("strob average is " + v);
}
}
```

The output is shown here:

```
Average is 3.0
Average is 3.3
```

Collection classes: Arrays, Vectors, Array list, Linked list, Hash set, Queues, Trees

Because **java.util** contains a wide array of functionality, it is quite large. Here is a list of its classes:

AbstractCollection	EventObject	Random
AbstractList	FormattableFlags	ResourceBundle
AbstractMap	Formatter	Scanner
AbstractQueue	GregorianCalendar	ServiceLoader (Added by Java SE 6.)
AbstractSequentialList	HashMap	SimpleTimeZone
AbstractSet	HashSet	Stack
ArrayDeque (Added by Java SE 6.)	Hashtable	StringTokenizer
ArrayList	IdentityHashMap	Timer
Arrays	LinkedHashMap	TimerTask
BitSet	LinkedHashSet	TimeZone
Calendar	LinkedList	TreeMap
Collections	ListResourceBundle	TreeSet
Currency	Locale	UUID
Date	Observable	Vector
Dictionary	PriorityQueue	WeakHashMap
EnumMap	Properties	
EnumSet	PropertyPermission	
EventListenerProxy	PropertyResourceBundle	

The ArrayList Class

The **ArrayList** class extends **AbstractList** and implements the **List** interface.

ArrayList is a generic class that has this declaration:

class ArrayList<E>

Here, **E** specifies the type of objects that the list will hold.

ArrayList supports dynamic arrays that can grow as needed. In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold. But, sometimes, you may not know until run time precisely how large an array you need. To handle this situation, the Collections Framework defines **ArrayList**.

In essence, an **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size. Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk.

Constructors in the ArrayList

1. ArrayList(): This constructor is used to build an empty array list. If we wish to create an empty ArrayList with the name arr, then, it can be created as:

```
ArrayList arr = new ArrayList();
```

2. ArrayList(Collection c): This constructor is used to build an array list initialized with the elements from the collection c. Suppose, we wish to create an ArrayList arr which contains the elements present in the collection c, then, it can be created as:

```
ArrayList arr = new ArrayList(c);
```

3. ArrayList(int capacity): This constructor is used to build an array list with initial capacity being specified. Suppose we wish to create an ArrayList with the initial size being N, then, it can be created as:

```
ArrayList arr = new ArrayList(N);
```

```
// Demonstrate ArrayList.

import java.util.*;
class ArrayListDemo {
public static void main(String args[]) {
// Create an array list.
ArrayList<String> al = new ArrayList<String>();
System.out.println("Initial size of al: " +
al.size());
// Add elements to the array list.
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
al.add(1, "A2");
System.out.println("Size of al after additions: " +
al.size());
// Display the array list.
System.out.println("Contents of al: " + al);
// Remove elements from the array list.
al.remove("F");
al.remove(2);
System.out.println("Size of al after deletions: " +
al.size());
System.out.println("Contents of al: " + al);
}
}
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

Point to Remember about ArrayList:

1. ArrayList is Underlined data Structure Resizable Array or Growable Array.
2. ArrayList Duplicates Are Allowed.
3. Insertion Order is Preserved.
4. Null insertion is possible.

Basis	Array	ArrayList
Definition	An **array** is a dynamically-created object. It serves as a container that holds the constant number of values of the same type. It has a contiguous memory location.	The **ArrayList** is a class of Java **Collections** framework. It contains popular classes like **Vector, HashTable**, and **HashMap**.
Static/ Dynamic	Array is **static** in size.	ArrayList is **dynamic** in size.
Resizable	An array is a **fixed-length** data structure.	ArrayList is a **variable-length** data structure. It can be resized itself when needed.
Initialization	It is mandatory to provide the size of an array while initializing it directly or indirectly.	We can create an instance of ArrayList without specifying its size. Java creates ArrayList of default size.
Performance	It performs **fast** in comparison to ArrayList because of fixed size.	ArrayList is internally backed by the array in Java. The resize operation in ArrayList slows down the performance.
Primitive/ Generic type	An array can store both **objects** and **primitives** type.	We cannot store **primitive** type in ArrayList. It automatically converts primitive type to object.
Iterating Values	We use **for** loop or **for each** loop to iterate over an array.	We use an **iterator** to iterate over ArrayList.
Type-Safety	We cannot use generics along with array because it is not a convertible type of array.	ArrayList allows us to store only **generic/ type, that's why it is type-safe.**
Length	Array provides a **length** variable which denotes the length of an array.	ArrayList provides the **size()** method to determine the size of ArrayList.
Adding Elements	We can add elements in an array by using the **assignment** operator.	Java provides the **add()** method to add elements in the ArrayList.
Single/ Multi-Dimensional	Array can be **multi-dimensional**.	ArrayList is always **single-dimensional**.

The LinkedList Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure. **LinkedList** is a generic class that has this declaration:

class LinkedList<E>

Here, **E** specifies the type of objects that the list will hold.

Constructors in the LinkedList:
In order to create a LinkedList, we need to create an object of the LinkedList class. The LinkedList class consists of various constructors that allow the possible creation of the list. The following are the constructors available in this class:

1. LinkedList(): This constructor is used to create an empty linked list. If we wish to create an empty LinkedList with the name ll, then, it can be created as:

```
LinkedList ll = new LinkedList();
```

2. LinkedList(Collection C): This constructor is used to create an ordered list that contains all the elements of a specified collection, as returned by the collection's iterator. If we wish to create a LinkedList with the name ll, then, it can be created as:

```
LinkedList ll = new LinkedList(C);
```

```
// Demonstrate LinkedList.

import java.util.*;
class LinkedListDemo {
public static void main(String args[]) {
// Create a linked list.
LinkedList<String> ll = new LinkedList<String>();
// Add elements to the linked list.
ll.add("F");
ll.add("B");
ll.add("D");
ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");

ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);
// Remove elements from the linked list.
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "+ ll);
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "+ ll);
// Get and set a value.
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
}
}
```

The output from this program is shown here:

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

```
// Java program to iterate the elements in an LinkedList
import java.util.*;
public class Demo {
	public static void main(String args[])
	{
		LinkedList<String> ll = new LinkedList<>();
		ll.add(“Shri");
		ll.add(“College");
		ll.add(1, “Ramdeobaba");

		// Using the Get method and the for loop
		for (int i = 0; i < ll.size(); i++) {

			System.out.print(ll.get(i) + " ");
		}
		System.out.println();
			// Using the for each loop
		for (String str : ll)
			System.out.print(str + " ");
	}
}
```

Output:

Shri Ramdeobaba College

Shri Ramdeobaba College

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the other elements are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data.	LinkedList is **better for manipulating** data.
5) The memory location for the elements of an ArrayList is contiguous.	The location for the elements of a linked list is not contagious.
6) Generally, when an ArrayList is initialized, a default capacity of 10 is assigned to the ArrayList.	There is no case of default capacity in a LinkedList. In LinkedList, an empty list is created when a LinkedList is initialized.

The HashSet Class

HashSet extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

class HashSet<E>
Here, **E** specifies the type of objects that the set will hold.

As most readers likely know, a hash table stores information by using a mechanism called **hashing**. In *hashing,* the informational content of a key is used to determine a unique value, called its *hash code.* The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically— you never see the hash code itself. Also, your code can't directly index the hash table. The advantage of hashing is that it allows the execution time of **add( )**, **contains( )**, **remove( )**, and **size( )** to remain constant even for large sets.

It is important to note that **HashSet** does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as **TreeSet**, is a better choice.

SN	Constructor	Description
1)	HashSet()	It is used to construct a default HashSet.
2)	HashSet(int capacity)	It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet.
3)	HashSet(int capacity, float loadFactor)	It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor.
4)	HashSet(Collection<? extends E> c)	It is used to initialize the hash set by using the elements of the collection c.

```
// Demonstrate HashSet.
import java.util.*;
class HashSetDemo {
public static void main(String args[]) {
// Create a hash set.
HashSet<String> hs = new HashSet<String>();
// Add elements to the hash set.
hs.add("B");
hs.add("A");
hs.add("D");
hs.add("E");
hs.add("C");
hs.add("F");
System.out.println(hs);
}
}
```

The following is the output from this program:

```
[D, A, F, C, B, E]
```

As explained, the elements are not stored in sorted order, and the precise output may vary.

```
// Java program to Demonstrate Working of HashSet Class
// Importing required classes
import java.util.*;
// Main class
// HashSetDemo
class GFG {
	public static void main(String[] args)
	{
		// Creating an empty HashSet
		HashSet<String> h = new HashSet<String>();
		// Adding elements into HashSet
		// using add() method
		h.add("India");
		h.add("Australia");
		h.add("South Africa");
		// Adding duplicate elements
		h.add("India");
		// Displaying the HashSet
		System.out.println(h);
		System.out.println("List contains India or not:"+ h.contains("India"));
		// Removing items from HashSet
		// using remove() method
		h.remove("Australia");
		System.out.println("List after removing Australia:"+ h);
```

```
    // Display message
            System.out.println("Iterating over list:");

            // Iterating over hashSet items
            Iterator<String> i = h.iterator();

            // Holds true till there is single element remaining
            while (i.hasNext())

                // Iterating over elements
                // using next() method
                System.out.println(i.next());
        }
    }
```

Output:
[South Africa, Australia, India]
List contains India or not: true
List after removing Australia:[South Africa, India]
Iterating over list:
South Africa
India

The important points about Java HashSet class are:

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.
- The initial default capacity of HashSet is 16, and the load factor is 0.75.

- The default load factor of HashMap used in java, for instance, is 0.75f (75% of the map size). That means **if we have a HashTable with an array size of 100, then whenever we have 75 elements stored, we will increase the size of the array to double of its previous size i.e. to 200 now, in this case**

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

Arrays

The **Arrays** class provides various methods that are useful when working with arrays. The following program illustrates how to use some of the methods of the **Arrays** class:

```
// Demonstrate Arrays

import java.util.*;
class ArraysDemo {
public static void main(String args[]) {

// Allocate and initialize array.
int array[] = new int[10];
for(int i = 0; i < 10; i++)
array[i] = -3 * i;

// Display, sort, and display the array.
System.out.print("Original contents: ");
display(array);
Arrays.sort(array);
System.out.print("Sorted: ");
display(array);

// Fill and display the array.
Arrays.fill(array, 2, 6, -1);
System.out.print("After fill(): ");
display(array);

// Sort and display the array.
Arrays.sort(array);
System.out.print("After sorting again: ");
display(array);
```

```
        // Binary search for -9.
System.out.print("The value -9 is at location ");
int index = Arrays.binarySearch(array, -9);
System.out.println(index);
}
static void display(int array[]) {
        for(int i: array)
        System.out.print(i + " ");
        System.out.println();
        }
}
```

The following is the output from this program:

```
Original contents: 0 -3 -6 -9 -12 -15 -18 -21 -24 -27
Sorted: -27 -24 -21 -18 -15 -12 -9 -6 -3 0
After fill(): -27 -24 -1 -1 -1 -1 -9 -6 -3 0
After sorting again: -27 -24 -9 -6 -3 -1 -1 -1 -1 0
The value -9 is at location 2
```

Vector

Vector implements a dynamic array. It is similar to **ArrayList**, but with two differences: **Vector** is synchronized, and it contains many legacy methods that are not part of the Collections Framework **Vector** is declared like this:

class Vector<E>
Here, **E** specifies the type of element that will be stored.

a default vector, which has an initial size of 10

```
	// Demonstrate various Vector operations.
import java.util.*;
class VectorDemo {
public static void main(String args[]) {
	// initial size is 3, increment is 2
Vector<Integer> v = new Vector<Integer>(3, 2);
System.out.println("Initial size: " + v.size());
System.out.println("Initial capacity: " +
v.capacity());
```

```
v.addElement(1);
v.addElement(2);
v.addElement(3);
v.addElement(4);
System.out.println("Capacity after four additions: " +
v.capacity());
v.addElement(5);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(6);
v.addElement(7);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(9);
v.addElement(10);
System.out.println("Current capacity: " +
v.capacity());
v.addElement(11);
v.addElement(12);
System.out.println("First element: " +
v.firstElement());
System.out.println("Last element: " + v.lastElement());
if(v.contains(3))
System.out.println("Vector contains 3.");

      // Enumerate the elements in the vector.

Enumeration vEnum = v.elements();
System.out.println("\nElements in vector:");
while(vEnum.hasMoreElements())
System.out.print(vEnum.nextElement() + " ");
System.out.println();
}
}
```

The output from this program is shown here:

```
Initial size: 0
Initial capacity: 3
Capacity after four additions: 5
Current capacity: 5
Current capacity: 7
Current capacity: 9
First element: 1
Last element: 12
Vector contains 3.
Elements in vector:
1 2 3 4 5 6 7 9 10 11 12
```

```
// Use an iterator to display contents.

Iterator<Integer> vItr = v.iterator();
System.out.println("\nElements in vector:");
while(vItr.hasNext())
System.out.print(vItr.next() + " ");
System.out.println();
```

You can also use a for-each **for** loop to cycle through a **Vector**, as the following version of the preceding code shows:

```
// Use an enhanced for loop to display contents.
System.out.println("\nElements in vector:");
for(int i : v)
System.out.print(i + " ");
System.out.println();
```

The TreeSet Class

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface. It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:
class TreeSet<E>
Here, **E** specifies the type of objects that the set will hold.

```
// Demonstrate TreeSet.
import java.util.*;
class TreeSetDemo {
public static void main(String args[]) {
        // Create a tree set.
        TreeSet<String> ts = new TreeSet<String>();
        // Add elements to the tree set.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
        System.out.println(ts);
        }
}
```

The output from this program is shown here:

```
[A, B, C, D, E, F]
```

As explained, because **TreeSet** stores its elements in a tree, they are automatically arranged in sorted order, as the output confirms.

Sr. No.	Key	Hash Set	Tree Set
1	Implementation	Hash set is implemented using HashTable	The tree set is implemented using a tree structure.
2	Null Object	HashSet allows a null object	The tree set does not allow the null object. It throws the null pointer exception.
3	Methods	Hash set use equals method to compare two objects	Tree set use compare method for comparing two objects.
4	Ordering	HashSet does not maintain any order	TreeSet maintains an object in sorted order

//Priority queue

```
   /*  PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
System.out.println(queue.getClass());*/
```

Module-IV (Teaching Hours - 8)

Introduction to streams, byte streams, character streams, file handling in Java, Serialization
Multithreading: Java Thread models, creating thread using runnable interface and extending Thread, thread priorities, Thread Synchronization, Inter-thread communications.

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information.

A stream is linked to a physical device by the Java I/O system.

All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.

This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.

Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network

for example. Java implements streams within class hierarchies defined in the **java.io** package.

Java defines two types of streams: byte and character.

Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data.

Character streams provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized.

Also, in some cases, character streams are more efficient than byte streams.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies.

At the top are two abstract classes: **InputStream** and **OutputStream**.

Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers.

Remember, to use the stream classes, you must import **java.io**.

The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement.

Two of the most important are **read( )** and **write( )**, which, respectively, read and write bytes of data.

Both methods are declared as abstract inside **InputStream** and **OutputStream**.

They are overridden by derived stream classes.

The Character Stream Classes

Character streams are defined by using two class hierarchies.

At the top are two abstract classes, **Reader** and **Writer**.

These abstract classes handle Unicode character streams. Java has several concrete subclasses of each of these.

The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read( )** and **write( )**, which read and write characters of data, respectively.

These methods are overridden by derived stream classes.

The Predefined Streams

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment.

For example, using some of its methods, you can obtain the current time and the settings of various properties associated with the system.

System also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**.

This means that they can be used by any other part of your program and without reference to a specific **System** object.

System.out refers to the standard output stream. By default, this is the console.
System.in refers to standard input, which is the keyboard by default.
System.err refers to the standard error stream, which also is the console by default.

However, these streams may be redirected to any compatible I/O device.

System.in is an object of type **InputStream**;

System.out and **System.err** are objects of type **PrintStream**.

These are byte streams, even though they typically are used to read and write characters from and to the console.

As you will see, you can wrap these within character based streams, if desired.

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists. Today, using a byte stream to read console input is still technically possible, but doing so is not recommended.

The preferred method of reading console input is to use a character-oriented stream, which makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from **System.in**.
To obtain a character based stream that is attached to the console, wrap **System.in** in a **BufferedReader** object.
BufferedReader supports a buffered input stream. Its most commonly used constructor is shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created.

Reader is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:
InputStreamReader(InputStream *inputStream*)

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream.* Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

Reading Characters

```
// Use a BufferedReader to read characters from the
console.
import java.io.*;
class BRRead {
public static void main(String args[]) throws IOException
{
      char c;
      BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));

      System.out.println("Enter characters, 'q' to quit.");
      // read characters
      do {
            c = (char) br.read();
            System.out.println(c);
            } while(c != 'q');
      }
}
```

Here is a sample run:

```
Enter characters, 'q' to quit.
123abcq
1
2
3
a
b
c
q
```

This output may look a little different from what you expected, because **System.in** is line buffered, by default. This means that no input is actually passed to the program until you press ENTER. As you can guess, this does not make **read( )** particularly valuable for interactive console input.

Reading Strings

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class.

```
// Read a string from console using a BufferedReader.

import java.io.*;
class BRReadLines {
	public static void main(String args[]) throws IOException
	{
		// create a BufferedReader using System.in
		BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
		String str;
		System.out.println("Enter lines of text.");
		System.out.println("Enter 'stop' to quit.");
		do {
			str = br.readLine();
			System.out.println(str);
		}while(!str.equals("stop"));
	}
}
```

File

Although most of the classes defined by **java.io** operate on streams, the **File** class does not.

It deals directly with files and the file system. That is, the **File** class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself. A **File** object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.

Files are a primary source and destination for data within many programs.

The following constructors can be used to create **File** objects:

File(String *directoryPath*)
File(String *directoryPath*, String *filename*)
File(File *dirObj*, String *filename*)
File(URI *uriObj*)

```
// Demonstrate File.
import java.io.File;
class FileDemo {
	static void p(String s) {
		System.out.println(s);
}
public static void main(String args[]) {
	File f1 = new File("/java/COPYRIGHT");
	p("File Name: " + f1.getName());
	p("Path: " + f1.getPath());
	p("Abs Path: " + f1.getAbsolutePath());
	p("Parent: " + f1.getParent());
	p(f1.exists() ? "exists" : "does not exist");
	p(f1.canWrite() ? "is writeable" : "is not writeable");
	p(f1.canRead() ? "is readable" : "is not readable");
	p("is " + (f1.isDirectory() ? "" : "not" + " a directory"));
	p(f1.isFile() ? "is normal file" : "might be a named pipe");
	p(f1.isAbsolute() ? "is absolute" : "is not absolute");
	p("File size: " + f1.length() + " Bytes");
	}
}
```

When you run this program, you will see something similar to the following:

```
File Name: COPYRIGHT
Path: /java/COPYRIGHT
Abs Path: /java/COPYRIGHT
Parent: /java
exists
is writeable
is readable
is not a directory
is normal file
is absolute
File size: 695 Bytes
```

FileInputStream

The **FileInputStream** class creates an **InputStream** that you can use to read bytes from a file.
Its two most common constructors are shown here:

FileInputStream(String *filepath*)
FileInputStream(File *fileObj*)

Either can throw a **FileNotFoundException**.

Here, *filepath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.
The following example creates two **FileInputStream**s that use the same disk file and each of the two constructors:

```
FileInputStream f0 = new FileInputStream("/autoexec.bat")



File f = new File("/autoexec.bat");
FileInputStream f1 = new FileInputStream(f);
```

Although the first constructor is probably more commonly used, the second allows us to closely examine the file using the **File** methods, before we attach it to an input stream.

When a **FileInputStream** is created, it is also opened for reading. **FileInputStream** overrides six of the methods in the abstract class **InputStream**.

The next example shows how to read a single byte, an array of bytes, and a subrange array of bytes. It also illustrates how to use **available( )** to determine the number of bytes remaining, and how to use the **skip( )** method to skip over unwanted bytes. The program reads its own source file, which must be in the current directory.

```
// Demonstrate FileInputStream.

import java.io.*;

class FileInputStreamDemo {
	public static void main(String args[]) throws IOException {
		int size;

		InputStream f = new FileInputStream("FileInputStreamDemo.java");

		System.out.println("Total Available Bytes: " + (size = f.available()));
		int n = size/40;
		System.out.println("First " + n + " bytes of the file one read() at a time");
		for (int i=0; i < n; i++)
		 {
			System.out.print((char) f.read());
		}
		System.out.println("\nStill Available: " + f.available());
		System.out.println("Reading the next " + n + " with one read(b[])");
		byte b[] = new byte[n];
```

```
if (f.read(b) != n)
{
        System.err.println("couldn't read " + n + " bytes.");
}
System.out.println(new String(b, 0, n));
System.out.println("\nStill Available: " + (size = f.available()));
System.out.println("Skipping half of remaining bytes with skip()");

f.skip(size/2);
System.out.println("Still Available: " + f.available());
System.out.println("Reading " + n/2 + " into the end of array");

if (f.read(b, n/2, n/2) != n/2)
 {
System.err.println("couldn't read " + n/2 + " bytes.");
}
System.out.println(new String(b, 0, b.length));
System.out.println("\nStill Available: " + f.available());
f.close();
}
}
```

Here is the output produced by this program:

```
Total Available Bytes: 1433
First 35 bytes of the file one read() at a time
// Demonstrate FileInputStream.
im
Still Available: 1398
Reading the next 35 with one read(b[])
port java.io.*;
class FileInputS
Still Available: 1363
Skipping half of remaining bytes with skip()
Still Available: 682
Reading 17 into the end of array
port java.io.*;
read(b) != n) {
S
Still Available: 665
This somewhat contrived example demonstrates how to
read three ways, to skip input, and
to inspect the amount of data available on a stream.
```

FileOutputStream

FileOutputStream creates an **OutputStream** that you can use to write bytes to a file. Its most commonly used constructors are shown here:

FileOutputStream(String *filePath*)
FileOutputStream(File *fileObj*)
FileOutputStream(String *filePath*, boolean *append*)
FileOutputStream(File *fileObj*, boolean *append*)

They can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, the file is opened in append mode.
Creation of a **FileOutputStream** is not dependent on the file already existing.
FileOutputStream will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

The following example creates a sample buffer of bytes by first making a **String** and then using the **getBytes( )** method to extract the byte array equivalent. It then creates three files. The first, **file1.txt**, will contain every other byte from the sample. The second, **file2.txt**, will contain the entire set of bytes. The third and last, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileOutputStream.
import java.io.*;
class FileOutputStreamDemo {
public static void main(String args[]) throws IOException {
        String source = "Now is the time for all good men\n"
        + " to come to the aid of their country\n"
        + " and pay their due taxes.";
        byte buf[] = source.getBytes();
        OutputStream f0 = new FileOutputStream("file1.txt");
        for (int i=0; i < buf.length; i += 2) {
                f0.write(buf[i]);
        }
        f0.close();
        OutputStream f1 = new FileOutputStream("file2.txt");
        f1.write(buf);
        f1.close();
        OutputStream f2 = new FileOutputStream("file3.txt");
        f2.write(buf,buf.length-buf.length/4,buf.length/4);
        f2.close();
        }
}
```

Here are the contents of each file after running this program. First, **file1.txt**:

Nwi h iefralgo e
t oet h i ftercuty n a hi u ae.

Next, **file2.txt**:

Now is the time for all good men
to come to the aid of their country
and pay their due taxes.

Finally, **file3.txt**:

nd pay their due taxes.

DataOutputStream and DataInputStream

DataOutputStream and **DataInputStream** enable you to **write or read primitive data** to or from a stream.

They implement the **DataOutput** and **DataInput** interfaces, respectively.
These interfaces define methods that convert primitive values to or from a sequence of bytes. These streams make it easy to store binary data, such as integers or floating-point values, in a file.

DataOutputStream extends **FilterOutputStream**, which extends **OutputStream**.
DataOutputStream defines the following constructor:

DataOutputStream(OutputStream *outputStream*)

Here, *outputStream* specifies the output stream to which data will be written.

DataOutputStream supports all of the methods defined by it superclasses. However, it is the methods defined by the **DataOutput** interface, which it implements, that make it interesting. **DataOutput defines methods that convert values of a primitive type into a byte sequence and then writes it to the underlying stream.** Here is a sampling of these methods:
final void writeDouble(double *value*) throws IOException
final void writeBoolean(boolean *value*) throws IOException
final void writeInt(int *value*) throws IOException
Here, *value* is the value written to the stream.

The following program demonstrates the use of **DataOutputStream** and **DataInputStream**:

```
import java.io.*;
class DataIODemo {
	public static void main(String args[]) throws IOException {
		FileOutputStream fout = new FileOutputStream("Test.dat");
		DataOutputStream out = new DataOutputStream(fout);
		out.writeDouble(98.6);
		out.writeInt(1000);
		out.writeBoolean(true);
		out.close();
		FileInputStream fin = new FileInputStream("Test.dat");
		DataInputStream in = new DataInputStream(fin);
		double d = in.readDouble();
		int i = in.readInt();
		boolean b = in.readBoolean();
		System.out.println("Here are the values: " +
		d + " " + i + " " + b);
		in.close();
	}
}
```

The output is shown here:

```
Here are the values: 98.6 1000 true
```

FileReader

The **FileReader** class creates a **Reader** that you can use to read the contents of a file. Its two most commonly used constructors are shown here:

FileReader(String *filePath*)
FileReader(File *fileObj*)

Either can throw a **FileNotFoundException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file.

The following example shows how to read lines from a file and print these to the standard output stream. It reads its own source file, which must be in the current directory.

```
// Demonstrate FileReader.
import java.io.*;
class FileReaderDemo {
	public static void main(String args[]) throws IOException {
		FileReader fr = new FileReader("FileReaderDemo.java");
		BufferedReader br = new BufferedReader(fr);
		String s;
		while((s = br.readLine()) != null) {
			System.out.println(s);
		}
		fr.close();
	}
}
```

FileWriter

FileWriter creates a **Writer** that you can use to write to a file. Its most commonly used constructors are shown here:

FileWriter(String *filePath*)
FileWriter(String *filePath*, boolean *append*)
FileWriter(File *fileObj*)
FileWriter(File *fileObj*, boolean *append*)

They can throw an **IOException**. Here, *filePath* is the full path name of a file, and *fileObj* is a **File** object that describes the file. If *append* is **true**, then output is appended to the end of the file.

Creation of a **FileWriter** is not dependent on the file already existing. **FileWriter** will create the file before opening it for output when you create the object. In the case where you attempt to open a read-only file, an **IOException** will be thrown.

The following example is a character stream version of an example shown earlier when **FileOutputStream** was discussed. This version creates a sample buffer of characters by first making a **String** and then using the **getChars( )** method to extract the character array equivalent.

It then creates three files. The first, **file1.txt**, will contain every other character from the sample. The second, **file2.txt**, will contain the entire set of characters. Finally, the third, **file3.txt**, will contain only the last quarter.

```
// Demonstrate FileWriter.
import java.io.*;
class FileWriterDemo {
	public static void main(String args[]) throws IOException {
		String source = "Now is the time for all good men\n"
		+ " to come to the aid of their country\n"
		+ " and pay their due taxes.";
		char buffer[] = new char[source.length()];
		source.getChars(0, source.length(), buffer, 0);
		FileWriter f0 = new FileWriter("file1.txt");
		for (int i=0; i < buffer.length; i += 2) {
			f0.write(buffer[i]);
		}
		f0.close();
		FileWriter f1 = new FileWriter("file2.txt");
		f1.write(buffer);
		f1.close();
		FileWriter f2 = new FileWriter("file3.txt");
		f2.write(buffer,buffer.length-buffer.length/4,buffer.length/4);
		f2.close();
	}
}
```

Serialization

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to a persistent storage area, such as a file.

At a later time, you may restore these objects by using the process of deserialization.

Assume that an object to be serialized has references to other objects, which, in turn, have references to still more objects. This set of objects and the relationships among them form a directed graph. There may also be circular references within this object graph. That is, object X may contain a reference to object Y, and object Y may contain a reference back to object X. Objects may also contain references to themselves. The object serialization and deserialization facilities have been designed to work correctly in these scenarios. If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.
Similarly, during the process of deserialization, all of these objects and their references are correctly restored.

An overview of the interfaces and classes that support serialization follows.

Serializable

Only an object that implements the **Serializable** interface can be saved and restored by the serialization facilities. The **Serializable** interface defines no members. It is simply used to indicate that a class may be serialized. **If a class is serializable, all of its subclasses are also serializable.**

A Serialization Example

The following program illustrates how to use object serialization and deserialization. It begins by instantiating an object of class **MyClass**. This object has three instance variables that are of types **String**, **int**, and **double**. This is the information we want to save and restore.

A **FileOutputStream** is created that refers to a file named “serial,” and an **ObjectOutputStream** is created for that file stream. The **writeObject( )** method of **ObjectOutputStream** is then used to serialize our object. The object output stream is flushed and closed.

A **FileInputStream** is then created that refers to the file named “serial,” and an **ObjectInputStream** is created for that file stream. The **readObject( )** method of **ObjectInputStream** is then used to deserialize our object. The object input stream is then closed.

Note that **MyClass** is defined to implement the **Serializable** interface. If this is not done, a **NotSerializableException** is thrown.

```
import java.io.*;

public class SerializationDemo {
	public static void main(String args[]) {
		// Object serialization
		try {
			MyClass object1 = new MyClass("Hello", -7, 2.7e10);
			System.out.println("object1: " + object1);
			FileOutputStream fos = new FileOutputStream("serial");
			ObjectOutputStream oos = new ObjectOutputStream(fos);
			oos.writeObject(object1);
			oos.flush();
			oos.close();
		}
		catch(IOException e) {
			System.out.println("Exception during serialization: " + e);
			System.exit(0);
}
```

```
// Object deserialization

try {
        MyClass object2;
        FileInputStream fis = new FileInputStream("serial");
        ObjectInputStream ois = new ObjectInputStream(fis);
        object2 = (MyClass)ois.readObject();
        ois.close();
        System.out.println("object2: " + object2);
}
catch(Exception e) {
        System.out.println("Exception during deserialization: " + e);
        System.exit(0);
        }
}
}
```

```
class MyClass implements Serializable {
	String s;
	int i;
	double d;
	public MyClass(String s, int i, double d) {
		this.s = s;
		this.i = i;
		this.d = d;
	}
public String toString() {
	return "s=" + s + "; i=" + i + "; d=" + d;
	}
}
```

This program demonstrates that the instance variables of **object1** and **object2** are identical.

The output is shown here:

```
object1: s=Hello; i=-7; d=2.7E10
object2: s=Hello; i=-7; d=2.7E10
```

Multithreading: Java Thread models, creating thread using runnable interface and extending Thread, thread priorities, Thread Synchronization, InterThread communications.

A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

Threads exist in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed.

Thread Priorities

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another. As an absolute value, a priority is meaningless; a higher-priority thread doesn't run any faster than a lower-priority thread if it is the only thread running. Instead, a thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch. The rules that determine when a context switch takes place are simple:

A thread can voluntarily relinquish control. This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.

A thread can be preempted by a higher-priority thread. In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For operating systems such as Windows, threads of equal priority are time-sliced automatically in round-robin fashion. For other types of operating systems, threads of equal priority must voluntarily yield control to their peers. If they don't, the other threads will not run.

Synchronization

Because multithreading introduces an asynchronous behavior to your programs, there must be a way for you to enforce synchronicity when you need it.

For example, if you want two threads to communicate and share a complicated data structure, such as a linked list, you need some way to ensure that they don't conflict with each other. That is, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements an elegant twist on an age-old model of inter process synchronization: the *monitor.*

You can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor, all other threads must wait until that thread exits the monitor. In this way, a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Most multithreaded systems expose monitors as objects that your program must explicitly acquire and manipulate. Java provides a cleaner solution. There is no class "Monitor"; instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables you to write very clear and concise multithreaded code, because synchronization support is built into the language.

Messaging

After you divide your program into separate threads, you need to define how they will communicate with each other.

When programming with most other languages, you must depend on the operating system to establish communication between threads. This, of course, adds overhead.

By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter a synchronized method on an object, and then **wait** there until some other thread explicitly **notifies** it to come out.

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the Thread class, its methods, and its companion interface, Runnable.

To create a new thread, your program will either extend Thread or implement the Runnable interface.

The Thread class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method	Meaning
getName	Obtain a thread's name.
getPriority	Obtain a thread's priority.
isAlive	Determine if a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread. Its general form is shown here:

static Thread currentThread( )

This method returns a reference to the thread in which It is called. Once you have a reference to the main thread, you can control it just like any other thread.

```
// Controlling the main Thread.
class CurrentThreadDemo {
	public static void main(String args[]) {
		Thread t = Thread.currentThread();
		System.out.println("Current thread: " + t);
		// change the name of the thread
		t.setName("My Thread");
		System.out.println("After name change: " + t);
		try {
			for(int n = 5; n > 0; n--) {
				System.out.println(n);
				Thread.sleep(1000);
			}
		} catch (InterruptedException e) {
		System.out.println("Main thread interrupted");
		}
	}
}
```

Here is the output generated by this program:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5
4
3
2
1
```

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

Runnable abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

public void run( )

Inside **run( )**, you will define the code that constitutes the new thread. It is important to understand that **run( )** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run( )** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run( )** returns.

After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**. In essence, **start( )** executes a call to **run( )**. The **start( )** method is shown here:

void start( )

```
// Create a second thread.
class NewThread implements Runnable {
        Thread t;
        NewThread() {
        // Create a new, second thread
        t = new Thread(this, "Demo Thread");
        System.out.println("Child thread: " + t);
        t.start(); // Start the thread
        }
        // This is the entry point for the second thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                                }
                        } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                        }
                System.out.println("Exiting child thread.");
        }
}
```

```
class ThreadDemo {
        public static void main(String args[]) {
                new NewThread(); // create a new thread
                try {
                for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
                } catch (InterruptedException e) {
                System.out.println("Main thread
                interrupted.");
                }
        System.out.println("Main thread exiting.");
        }
}
```

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.
```

Extending Thread

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class. The extending class must override the **run( )** method, which is the entry point for the new thread. It must also call **start( )** to begin execution of the new thread.

```
// Create a second thread by extending Thread
class NewThread extends Thread {
    NewThread() {
        // Create a new, second thread
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
        }
        // This is the entry point for the second
        thread.
        public void run() {
        try {
        for(int i = 5; i > 0; i--) {
            System.out.println("Child Thread: " + i);
            Thread.sleep(500);
            }
        } catch (InterruptedException e) {
        System.out.println("Child interrupted.");
    }
    System.out.println("Exiting child thread.");
    }
}
```

```
class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println("Main Thread: " + i);
        Thread.sleep(1000);
        }
    }
    catch (InterruptedException e) {
        System.out.println("Main thread
    interrupted.");
    }
        System.out.println("Main thread exiting.");
    }
}
```

Creating Multiple Threads

```
// Create multiple threads.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        NewThread(String threadname) {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start(); // Start the thread
        }
        // This is the entry point for thread.
        public void run() {
        try {
        for(int i = 5; i > 0; i--) {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
}
catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
        System.out.println(name + " exiting.");
        }
}
class MultiThreadDemo {
public static void main(String args[]) {
        new NewThread("One"); // start threads
        new NewThread("Two");
        new NewThread("Three");
        try {
        // wait for other threads to end
        Thread.sleep(10000);
        } catch (InterruptedException e) {
        System.out.println("Main thread
Interrupted");
}
        System.out.println("Main thread exiting.");
}
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

Using isAlive( ) and join( )

As mentioned, often you will want the main thread to finish last. In the preceding examples, this is accomplished by calling **sleep( )** within **main( )**, with a long enough delay to ensure that all child threads terminate prior to the main thread. However, this is hardly a satisfactory solution, and it also raises a larger question: How can one thread know when another thread has ended? Fortunately, **Thread** provides a means by which you can answer this question.

Two ways exist to determine whether a thread has finished. First, you can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.
While **isAlive( )** is occasionally useful, the method that you will more commonly use **to wait for a thread to finish is called join( )**, shown here:

final void join( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread *joins* it. Additional forms of **join( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Using isAlive( ) and join( )

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
      String name; // name of thread
      Thread t;
      NewThread(String threadname) {
            name = threadname;
            t = new Thread(this, name);
            System.out.println("New thread: " + t);
            t.start(); // Start the thread
      }
      // This is the entry point for thread.
      public void run() {
      try {
      for(int i = 5; i > 0; i--) {
            System.out.println(name + ": " + i);
            Thread.sleep(1000);
      }
}
catch (InterruptedException e) {
      System.out.println(name + " interrupted.");
      }
      System.out.println(name + " exiting.");
      }
}
class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");

System.out.println("Thread One is alive: "+ ob1.t.isAlive());
System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
```

```
// wait for threads to finish
try {
      System.out.println("Waiting for threads to finish.");
      ob1.t.join();
      ob2.t.join();
      ob3.t.join();
      }
      catch (InterruptedException e) {
      System.out.println("Main thread Interrupted");
      }

      System.out.println("Thread One is alive: “ + ob1.t.isAlive());
      System.out.println("Thread Two is alive: “ + ob2.t.isAlive());
      System.out.println("Thread Three is alive: “ + ob3.t.isAlive());
      System.out.println("Main thread exiting.");
      }
}
```

Sample output from this program is shown here. (Your output may vary based on processor speed and task load.)

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
```

```
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

As you can see, after the calls to **join( )** return, the threads have stopped executing.

Thread Priorities

To set a thread's priority, use the **setPriority( )** method, which is a member of **Thread**.

This is its general form:

final void setPriority(int *level*)

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority( )** method of **Thread**, shown here:

final int getPriority( )

```
// Demonstrate thread priorities.
class clicker implements Runnable {
    long click = 0;
    Thread t;
    private volatile boolean running = true;
    public clicker(int p) {
    t = new Thread(this);
    t.setPriority(p);
    }
    public void run() {
        while (running) {
            click++;
        }
    }
    public void stop() {
        running = false;
    }
    public void start() {
        t.start();
    }
}
class HiLoPri {
public static void main(String args[]) {
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
lo.start();
hi.start();
try {
    Thread.sleep(10000);
}
catch (InterruptedException e) {
    System.out.println("Main thread interrupted.");
}
lo.stop();
hi.stop();
// Wait for child threads to terminate.
try {
    hi.t.join();
    lo.t.join();
}
```

```
catch (InterruptedException e) {
        System.out.println("InterruptedException caught");
}
        System.out.println("Low-priority thread: " + lo.click);
        System.out.println("High-priority thread: " + hi.click);
        }
}
```

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the **resource will be used by only one thread at a time**. The process by which this is achieved is called *synchronization.* As you will see, Java provides unique, language-level support for it.

Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex.* Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can re-enter the same monitor if it so desires.

```
// This program is not synchronized.
class Callme {
	void call(String msg) {
		System.out.print("[" + msg);
	try {
		Thread.sleep(1000);
	} catch(InterruptedException e) {
		System.out.println("Interrupted");
		}
	System.out.print("]");
	}
}
class Caller implements Runnable {
	String msg;
	Callme target;
	Thread t;
	public Caller(Callme targ, String s) {
	target = targ;
	msg = s;
	t = new Thread(this);
	t.start();
}
public void run() {
	target.call(msg);
	}
}
class Synch {
public static void main(String args[]) {
	Callme target = new Callme();
	Caller ob1 = new Caller(target, "Hello");
	Caller ob2 = new Caller(target, "Synchronized");
	Caller ob3 = new Caller(target, "World");
	// wait for threads to end
	try {
		ob1.t.join();
		ob2.t.join();
		ob3.t.join();
	} catch(InterruptedException e) {
		System.out.println("Interrupted");
		}
	}
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]]]
```

To fix the preceding program, you must *serialize* access to **call( )**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call( )**'s definition with the keyword **synchronized**, as shown here:

```
class Callme {
synchronized void call(String msg) {
...
```

This prevents other threads from entering **call( )** while another thread is using it. After **synchronized** has been added to **call( )**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, non synchronized methods on that instance will continue to be callable.

Interthread Communication

Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context.

- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**.
- **notify( )** wakes up a thread that called **wait( )** on the same object.
- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object.

These methods are declared within **Object**, as shown here:

```
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )
```

Consider the following sample program that incorrectly implements a simple form of the producer/ consumer problem.

It consists of four classes:
Q, the queue that you're trying to synchronize;
Producer, the threaded object that is producing queue entries;
Consumer, the threaded object that is consuming queue entries; and
PC, the tiny class that creates the single **Q**, **Producer**, and **Consumer**.

// An incorrect implementation of a producer and consumer.

```
class Q {
	int n;
	synchronized int get() {
		System.out.println("Got: " + n);
		return n;
	}
	synchronized void put(int n) {
		this.n = n;
		System.out.println("Put: " + n);
	}
}
class Producer implements Runnable {
	Q q;
	Producer(Q q) {
		this.q = q;
		new Thread(this, "Producer").start();
	}
	public void run() {
		int i = 0;
		while(true) {
		q.put(i++);
		}
	}
}
class Consumer implements Runnable {
	Q q;
	Consumer(Q q) {
		this.q = q;
		new Thread(this, "Consumer").start();
		}
		public void run() {
		while(true) {
			q.get();
		}
	}
}
```

```
class PC {
public static void main(String args[]) {
        Q q = new Q();
        new Producer(q);
        new Consumer(q);
        System.out.println("Press Control-C to stop.");
        }
}
```

Although the **put( )** and **get( )** methods on **Q** are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put: 1
Got: 1
Got: 1
Got: 1
Got: 1
Got: 1
Put: 2
Put: 3
Put: 4
Put: 5
Put: 6
Put: 7
Got: 7
```

```
// A correct implementation of a producer and consumer.
class Q {
    int n;
    boolean valueSet = false;

    synchronized int get() {
    while(!valueSet)
    try {
        wait();
    }
    catch(InterruptedException e) {
    System.out.println("InterruptedException caught");
    }
    System.out.println("Got: " + n);
    valueSet = false;
    notify();
    return n;
}
```

```
synchronized void put(int n) {
while(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
}
}
```

```
class Producer implements Runnable {
Q q;
Producer(Q q) {
	this.q = q;
	new Thread(this, "Producer").start();
}
public void run() {
	int i = 0;
	while(true) {
		q.put(i++);
	}
	}
}
```

```
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
	this.q = q;
	new Thread(this, "Consumer").start();
}
public void run() {
	while(true) {
	q.get();
	}
	}
}
class PCFixed {
public static void main(String args[]) {
	Q q = new Q();
	new Producer(q);
	new Consumer(q);
	System.out.println("Press Control-C to stop.");
	}
}
```

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```