DEPARTMENT OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

Program: BCA

SUBJECT NAME: JAVA PROGRAMMING
SUBJECT CODE: 23BCA2C03
SEM: II

MODULE-2

- **Objects and classes: Defining classes,Objects and methods,Constructors.**
- **thiskeyword,Accessing objects via reference variables,**
- **Accessor and Mutator methods,Access odifiers,**
- **Constructors and Method Overloading,**
- **Static variables,constants and methods,**
- **Passing objects to methods,Array of objects,**
- **Packages,Wrapper classes,**
- **Class relations:dependency,association.**

Introducing Classes

The class is at the core of Java. It is the logical construct upon which the entire Java language is built because it defines the shape and nature of an object. As such, the class forms the basis for object-oriented programming in Java.

Class Fundamentals

Classes have been used since the beginning of this book. However, until now, only the most rudimentary form of a class has been shown. The classes created in the preceding chapters primarily exist simply to encapsulate the main( ) method, which has been used to demonstrate the basics of the Java syntax.

The General Form of a Class

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

```
class classname
{
type instance-variable1;

type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list)
{
// body of method
}
type methodname2(parameter-list)
{
// body of method
}
// ...
type methodnameN(parameter-list)
 {
// body of method
}
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

A Simple Class

Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods

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

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

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

Thus, everyBox object will contain its own copies of the instance variables width, height, and depth. To access these variables, you will use the dot (.) operator. The dot operator links the name of the object with the name of an instance variable. For example, to assign the width variable of mybox the value 100, you would use the following statement:

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

This statement tells the compiler to assign the copy of width that is contained within the mybox object the value of 100. In general, you use the dot operator to access both the instance variables and the methods within an object.

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

Declaring Objects

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.

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
```

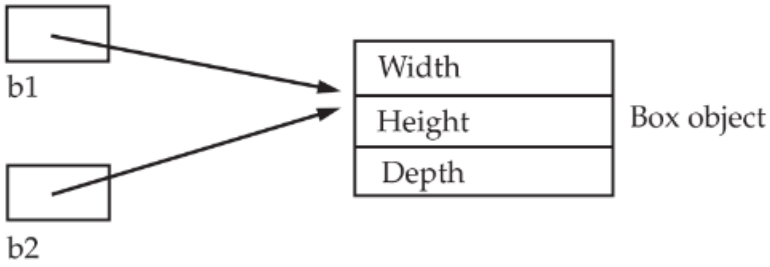
The first line declares mybox as a reference to an object of type Box. At this point, mybox does not yet refer to an actual object. The next line allocates an object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds, in essence, the memory address of the actualBox object.

Assigning Object Reference Variables

Object reference variables act differently than you might expect when an assignment takes place.

```
Box b2 = b1;
```

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.

Introducing Methods

This is the general form of a method:

```
type name(parameter-list)
{
// body of method
}
```

Here, type specifies the type of data returned by the method. This can be any valid type, including class types that you create. If the method does not return a value, its return type must be void. The name of the method is specified by name. This can be any legal identifier other than those already used by other items within the current scope. The parameter-list is asequence of type and identifier pairs separated by commas. Parameters are essentially variables that receive the value of the arguments passed to the method when it is called. If the method has no parameters, then the parameter list will be empty.
Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

```
return value;
```

Here, value is the value returned.
In the next few sections, you will see how to create various types of methods, including those that take parameters and those that return values.

Adding a Method to 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);
}
}
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();
}
}
mybox1.volume();
mybox2.volume();
```

The first line here invokes the volume( ) method on mybox1. That is, it calls volume( )relative to the mybox1 object, using the object's name followed by the dot operator. Thus, the call to mybox1.volume( ) displays the volume of the box defined by mybox1, and the call to mybox2.volume( ) displays the volume of the box defined by mybox2. Each time volume( ) is invoked, it displays the volume for the specified box.

Returning a Value

A better way to implement volume( ) is to have it compute the volume of the box and return the result to the caller. The following example, an improved version of the preceding program, does just that:

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

As you can see, when volume( ) is called, it is put on the right side of an assignment statement. On the left is a variable, in this case vol, that will receive the value returned by volume( ). Thus, after

```
vol = mybox1.volume();
```

executes, the value of mybox1.volume( ) is 3,000 and this value then is stored in 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.

One more point: The preceding program can be written a bit more efficiently because there is actually no need for the vol variable. The call to volume( ) could have been used in the println( ) statement directly, as shown here:
System.out.println("Volume is" + mybox1.volume());
generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations. To illustrate this point, let's use a very simple example. Here is a method that returns the square of the number 10:

```
int square()
{
return 10 * 10;

}
```

Adding a Method That Takes Parameters

While this method does, indeed, return the value of 10 squared, its use is very limited. However, if you modify the method so that it takes a parameter, as shown next, then you can make square( ) much more useful.

```
int square(int i)
{
return i * i;
}
```

Now, square( ) will return the square of whatever value it is called with. That is, square( ) is now a general-purpose method that can compute the square of any integer value, rather than just 10.
Here is an example:

```
int x, y;
x = square(5); // x equals 25
x = square(9); // x equals 81
```

Accessing objects via reference variable

In Java, you can access objects via reference variables. In Java, all non-primitive data types are reference types, which means that when you create an object, you're actually creating a reference to that object. Here's a brief explanation of how it works:

Creating Objects: When you create an object in Java, you use the new keyword to allocate memory for the object, and a reference variable is used to refer to that object. For example:

MyClass MyClass obj = new MyClass();

Here, obj is a reference variable of type MyClass that points to the newly created object.

Accessing Object Members: You can access the members (fields and methods) of the object using the reference variable. For example:
Accessing a field
int value = obj.someField;

Calling a method
obj.someMethod();
Here, someField is a field of the object, and someMethod() is a method of the object.

Passing Objects as Parameters: You can pass objects as parameters to methods. When you pass an object as an argument, you're actually passing the reference to the object. Any changes made to the object inside the method will affect the original object outside the method.

```
void modifyObject(MyClass obj)
{
  obj.someField = 42;
}

// Usage
MyClass myObject = new MyClass();
modifyObject(myObject);

// The value of someField in myObject is now 42
```

Constructors

A constructor initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method. Once defined, the constructor is automatically called when the object is created, before the new operator completes. Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.It is the constructor's job to initialize the internal state of an object so that the code creating an instance will have a fully initialized, usable object immediately.

```
/* 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
```

As you can see, both mybox1 and mybox2 were initialized by the Box( ) constructor when they were created. Since the constructor gives all boxes the same dimensions, 10 by10 by 10, both mybox1 and mybox2 will have the same volume. The println( ) statement inside Box( ) is for the sake of illustration only. Most constructors will not display anything. They will simply initialize an object. Before moving on, let's reexamine the new operator. As you know, when you allocate a object, you use the following general form:

```
class-var = new classname ( );
```

Now you can understand why the parentheses are needed after the class name. What isactually happening is that the constructor for the class is being called. Thus, in the line

```
Box mybox1 = new Box();
```

new Box( ) is calling the Box( ) constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. This is why the preceding line of code worked in earlier versions of Box that did not define a constructor. The default constructor automatically initializes all instance variables to their default values, which are zero, null, and false, for numeric types, reference types, and boolean, respectively. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors

While the Box( ) constructor in the preceding example does initialize a Box object, it is not very useful—all boxes have the same dimensions. What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor. As you can probably guess, this makes it much more useful. For example, the following version of Box defines a parameterized constructor that sets the dimensions of a box as specified by those parameters. Pay special attention to how Box objects are created./* 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
As you can see, each object is initialized as specified in the parameters to its constructor.
For example, in the following line,

Box mybox1 = new Box(10, 20, 15);
the values 10, 20, and 15 are passed to the Box( ) constructor when new creates the object.
Thus, mybox1's copy of width, height, and depth will contain the values 10, 20, and 15,respectively.

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

```
<class_name>(){}
```

Java Program to create and call a default constructor

```
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
        /main method
public static void main(String args[]){
/calling a default constructor
Bike1 b=new Bike1();
}
```

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

Program to initialize the values from one object to another object.

```
class Student6
{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
    name = n;
    }

    //constructor to initialize another object
```

```
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display()
{
System.out.println(id+" "+name);
}
 public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
    s1.display();
    s2.display();
   }
}
```

Output:

111 Karan

111 Karan

Difference between constructor and method in Java

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Javadefines 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. **this keyword in Java** is a reference variable that refers to the current object of a method or a constructor. The main purpose of using this keyword in **Java** is to remove the confusion between class attributes and parameters that have same names.

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;
}
```

This version of Box( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object. While it is redundant in this case

this is useful in other contexts, one of which is explained in the next section.

Instance Variable Hiding

As you know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable hides the instance variable. This is why width, height, and depth were not used as the names of the parameters to the Box( ) constructor inside the Box class. If they had been, then width, for example, would have referred to the formal parameter, hiding the instance variable width. While it is usually easier to simply use different names, there is another way around this situation. Because this lets you refer directly to the object, you can use it to resolve any namespace collisions that might occur between instance variables and local variables. For example, here is another version of Box( ), which uses width, height, and depth for parameter names and then uses this to access the instance variables by the same name:

```
// Use this to resolve name-space collisions.
Box(double width, double height, double depth) {
this.width = width;
this.height = height;
this.depth = depth;
```

Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When 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. If you have never used a language that allows the overloading of methods, then the concept may seem strange at first. But as you will see, method overloading is one of Java's most exciting and useful features.

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.

Here is a simple example that illustrates method overloading:

```
// 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
```

As you can see, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution. When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases, Java's automatic type conversions can play a role in overload resolution.

For example, consider the following program:

```
// Automatic type conversions apply to overloading.
```

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

}
// Overload test for a double parameter
void test(double a)
{
System.out.println("Inside test(double) a: " + a);
}
}
class Overload {
public static void main(String args[]) {
OverloadDemo ob = new OverloadDemo();
int i = 88;
ob.test();
ob.test(10, 20);
ob.test(i); // this will invoke test(double)
ob.test(123.2); // this will invoke test(double)
}
}
```

This program generates the following output:

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

As you can see, this version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call. Therefore, after test (int) is not found, Java elevates i to double and then calls test(double). Of course, if test (int) had been defined, it would have been called instead.

Java will employ its automatic type conversions only if no exact match is found. Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name.

For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the

absolute value of a long integer, and fabs( ) returns the absolute value of a floating-point value. Since C does not support overloading, each function has its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called abs( ).This method is overloaded by Java's Math class to handle all numeric types. Java determines which version of abs( ) to call based upon the type of argument.

Overloading Constructors

In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception. To understand why, let's return to the Box class developed in the preceding chapter. Following is the latest version of 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;
}
}
```

As you can see, the Box( ) constructor requires three parameters. This means that alldeclarations of Box objects must pass three arguments to the Box( ) constructor. Fo rexample, the following statement is currently invalid:

```
Box ob = new Box();
```

Since Box( ) requires three arguments, it's an error to call it without them. This raises some important questions. What if you simply wanted a box and did not care (or know)what its initial dimensions were? Or, what if you want to be able to initialize a cube by specifying only one value that would be used for all three dimensions? As the Box class is currently written, these other options are not available to you.

Fortunately, the solution to these problems is quite easy: simply overload the Box constructor so that it handles the situations just described. Here is a program that contains an improved version of Box that does just that:

```
/* 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 equalTo(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.equalTo(ob2));
System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
}
}
```

This program generates the following output:
ob1 == ob2: true
ob1 == ob3: false

As you can see, the equalTo( ) method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed. If they contain the same values, then the method returns true. Otherwise, it returns false. Notice that the parameter o in equalTo( ) specifies Test as its type. Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.

Introducing Access Control

Java's access modifiers 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 public, 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 modifier. It is called by code that is outside the program—that is, by the Java run-time system. Whenno access modifier is used, then by default the member of a class is public within its own package, but cannot be accessed outside of its package.

To understand the effects of public and private access, consider the following program:

```
/* 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( ).

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access modifiers. For example, you already know that access to a private member of a class is granted only to other members of that class. Packages add another dimension to access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, sub classes, and packages.

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 modifiers, private, public, and protected, provide a variety of ways to produce the many levels of access required by these categories. 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 isthe 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.

	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

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 directly call other static methods.
• They can only directly 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*4or 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. For example, if you wish to call a static method from outside its class, you can do so using the following general form:

```
classname.method( )
```

Here, class name 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
```

Packages

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 exposed only 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 belong 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 from the one above it by use of a period. The general form of a multileveled package statement 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

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 MyPackdirectory. Then, try executing the AccountBalance

class, using the following
java MyPack.AccountBalance

Remember, you will need to be in the directory above MyPack when you execute this command. (Alternatively, you can use one of the other two options described in the preceding section to specify the path MyPack.)
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.

Importing Packages

Given that packages exist and are a good mechanism for compartmentalizing diverse classesfrom each other, it is easy to see why all of the built-in Java classes are stored in packages.There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. The import statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the import statement will save a lot of typing.

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 | *);

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit class name or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:
import java.util.Date;
import java.io.*;

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.*;

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile

time error and have to explicitly name the class specifying its package.It must be emphasized that the import statement is optional. Any place you use a classname, you can use its fully qualified name, which includes its full package hierarchy
Forexample, this fragment uses an import statement:

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

The same example without the import statement looks like this:

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

We'll create a package named com.example with two classes: MainClass and MyClass.

```
package com.example;
public class MainClass
 {
public static void main(String[] args)
 {
System.out.println("Hello from MainClass!");
 // Creating an instance of MyClass
MyClass myObject = new MyClass();
myObject.displayMessage();
 }
}
package com.example;
 public class MyClass
{
 public void displayMessage()
{
 System.out.println("Hello from MyClass!");
}
}
```

Open a terminal or command prompt, navigate to the myproject directory, and compile the code:

```
javac src/com/example/MainClass.java
```

This will compile both MainClass and MyClass.
Run the compiled program:

```
java com.example.MainClass
```

You should see the output:

```
Hello from MainClass!
Hello from MyClass!
```

Accessor methods and Mutator methods

Accessor methods and mutator methods, often referred to as getters and setters, are two types of methods commonly used in object-oriented programming to access and modify the private fields of a class.

Accessor Methods (Getters):

Purpose: Accessor methods are used to retrieve the values of private fields (attributes) in a class.
Naming Convention: Typically named with the prefix "get" followed by the name of the attribute they retrieve.
Return Type: They return the value of the attribute.

Example:

```
public class MyClass
{
private int myNumber;
// Accessor method for myNumber
public int getMyNumber()
{
return myNumber;
}
}
```

Mutator Methods (Setters):

Purpose: Mutator methods are used to modify the values of private fields in a class.
Naming Convention: Typically named with the prefix "set" followed by the name of the attribute they modify.
Parameters: They take a parameter representing the new value to set.
Return Type: They usually have a void return type, as they are primarily used to modify the internal state.

Example:

```
public class MyClass
{
private int myNumber;
 // Mutator method for myNumber
public void setMyNumber(int newValue)
 { myNumber = newValue;
}
}
```

Type Wrappers

Java provides type wrappers, which are classes that encapsulate a primitive type within an object. The type wrapper classes are described in detail in Part II, but they are introduced here because they

relate directly to Java's autoboxing feature.The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

Character

Character is a wrapper around a char. The constructor for Character is
Character(char ch)
Here, ch specifies the character that will be wrapped by the Character object being created.To obtain the char value contained in a Character object, call charValue( ), shown here:
char charValue( )
It returns the encapsulated character.

Boolean

Boolean is a wrapper around boolean values. It defines these constructors:
Boolean(boolean boolValue)
Boolean(String boolString)
In the first version, boolValue must be either true or false. In the second version, if boolStringcontains the string "true" (in uppercase or lowercase), then the new Boolean object will betrue. Otherwise, it will be false.To obtain a boolean value from a Boolean object, use booleanValue( ), shown here:
boolean booleanValue( )
It returns the boolean equivalent of the invoking object.

The Numeric Type Wrappers

By far, the most commonly used type wrappers are those that represent numeric values.These are Byte, Short, Integer, Long, Float, and Double. All of the numeric type wrappers inherit the abstract class Number. Number declares methods that return the value of anobject in each of the different number formats. These methods are shown here:
byte byteValue( )
double doubleValue( )
float floatValue( )
int intValue( )
long longValue( )
short shortValue( )

For example, doubleValue( ) returns the value of an object as a double, floatValue( )returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.
All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value. For example, here are the constructors defined for Integer:
Integer(int num)
Integer(String str)
If str does not contain a valid numeric value, then a NumberFormatException is thrown. All of the type wrappers override to String( ). It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to println( ), for example, without having to convert it into its primitive type.

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as

autoboxing,

Wrapper class Example: Primitive to Wrapper
//Java program to convert primitive into objects

```
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;

//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj
```

Array of Objects

Java Array Of Objects, as defined by its name, stores an array of objects. The array elements store the location of the reference variables of the object.
Syntax:
Class obj[]= new Class[array_length]

Example:

```
Class ObjectArray
{
public static void main(String args[])
{
Account obj[] = new Account[2] ;
//obj[0] = new Account();
obj[1] = new Account();
obj[0].setData(1,2);
obj[1].setData(3,4);
System.out.println("For Array Element 0");
obj[0].showData();
System.out.println("For Array Element 1");
obj[0].showData();
System.out.println("For Array Element 1");
obj[1].showData();
}
}
class Account
{ int a; int b;
public void setData(int c,int d)
{
a=c; b=d;
 }
 public void showData()
{ System.out.println("Value of a ="+a);
System.out.println("Value of b ="+b);
}
}
```

Output:

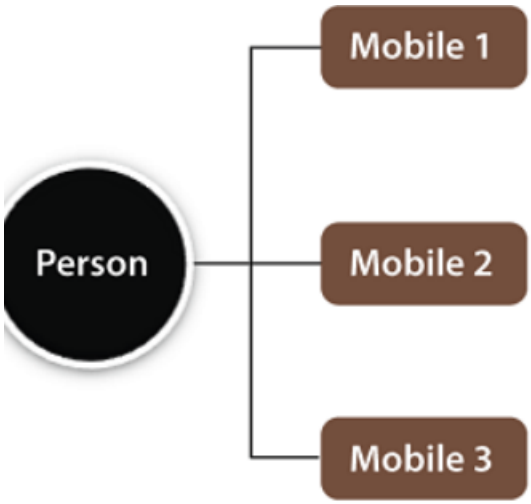
```
For Array Element 0
 Value of a =1
Value of b =2
 For Array Element 1
Value of a =3
Value of b =4
```

CLASS RELATIONSHIPS

Dependency
In object-oriented programming, a dependency between classes occurs when one class relies on another class, typically through method parameters or local variables.

Example: Dependency between Library and Book classes
In this example, the Library class has a method that depends on the Book class to perform a book checkout operation.

Association
Association in object-oriented programming represents a relationship between two or more classes. It is a way of connecting classes and can be classified into various types, such as one-to-one, one-to-many, and many-to-many associations.
.



A person can have only one passport. It defines the one-to-one.If we talk about the Association between a College and Student, a College can have many students. It defines the one-to-manyA state can have several cities, and those cities are related to that single state. It defines the many-to-oneA single student can associate with multiple teachers, and multiple students can also be associated with a single teacher. Both are created or deleted independently, so it defines the many-to-many.
Example: Association between Student and Course classes
Consider two classes: Student and Course. A student can be associated with multiple courses, and a course can have multiple students. This is an example of a many-to-many association

Aggregation

Aggregation in Java is a type of class relationship where one class contains an object of another class,

but the contained object has an independent lifecycle and can exist outside the containing class. Aggregation represents a "has-a" relationship.
EXAMPLE of aggregation between a University class and a Department class

Simple Example of Aggregation