**UNIT-2**

**CONTENTS**

**A Closer Look into Methods and Classes:** Controlling Access to Class Members, passing objects to methods, passing arguments, Returning Objects, Method Overloading, Overloading Constructors, Understanding Static, Variable-Length Arguments.

**A CLOSER LOOK INTO METHODS AND CLASSES**

**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.
* 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

**Here, 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 Overload1 {

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

* 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.

**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.

/\* 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

* 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.
* One of the most common uses of object parameters involves constructors.
* // 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);

}

}

**A CLOSER LOOK AT ARGUMENT PASSING**

* In general, there are two ways that a computer language can pass an argument to a subroutine. The first way is ***call-by-value*.** This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
* The second way an argument can be passed is ***call-by-reference*.** In this approach, a reference to an argument (not the value of the argument) is passed tothe parameter. Inside the subroutine, this reference is used to access the actual argumentspecified in the call. This means that changes made to the parameter will affect theargument used to call the subroutine.
* Java uses call-by-valueto pass all arguments, the precise effect differs between whether a primitive type or areference type is passed.When you pass a primitive type to a method, it is passed by value. Thus, a copy of theargument is made, and what occurs to the parameter that receives the argument has noeffect outside the method.
* For example, consider the following program:

// 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

* 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.

**NOTE:** When you pass an object to a method, the situation changes dramatically, because **objects are passed by what is effectively call-by-reference**.

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 act as if they 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 through their references.

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 PassObjRef {

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

As you can see, in this case, the actions inside **meth( )** have affected the object used as an

argument.

*NOTE: When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.*

**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

**INTRODUCING ACCESS CONTROL**

Encapsulation links data with the code that manipulates it. However, encapsulation provides another important attribute: *access control*. Through encapsulation, you can control what parts of a program can access the members of a class. By controlling access, you can prevent misuse. For example, allowing access to data only through a well-defined set of methods, you can prevent the misuse of that data. Thus, when correctly implemented, a class creates a “black box” which may be used, but the inner workings of which are not open to tampering.

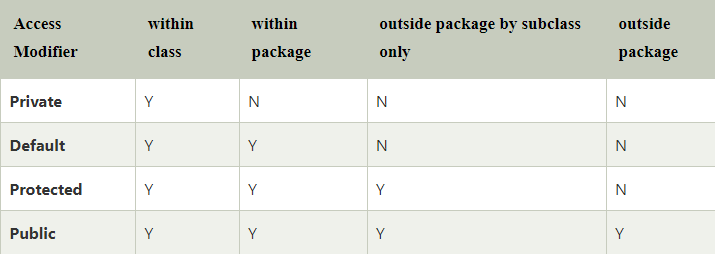
As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor, variable, method, or data member. There are four types of access modifiers available in java:

**Default – (No keyword required)** the access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

**Private:** The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

**Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

**Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.



How a member can be accessed is determined by the *access modifier* attached to its declaration. Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages. (A *package* is, essentially, a grouping of classes.)

Java’s access modifiers are **public**, **private**, and **protected**. Java also defines a default access level. **Protected** applies only when inheritance is involved.

Let’s begin by defining **public** and **private**. 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. When no 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.

An access modifier 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) { //...

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( )**. If you were to remove the comment symbol from the beginning of the following line,

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

Program on private

package p1;

class A

{

private void display()

{

System.out.println("Hello world");

}

}

class B

{

public static void main(String args[])

{

A obj = new A();

obj.display();

}

}

Output=error

Program on Protected

package p1;

public class A

{

protected void display()

{

System.out.println("Helloworld");

}

}

package p2;

import p1.\*;

class B extends A

{

public static void main(String args[])

{

B obj = new B();

obj.display();

}

}

Output:Helloworld

Program on public:

ckage p1;

public class A

{

public void display()

{

System.out.println("Helloworld");

}

}

package p2;

import p1.\*;

class B {

public static void main(String args[])

{

A obj = new A;

obj.display();

}

}

Output:Helloworld

**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. (The keyword **super** relates to inheritance)

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. 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, *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 objectreference 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

**VARARGS: VARIABLE-LENGTH ARGUMENTS**

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

Situations that require that a variable number of arguments be passed to a method are not unusual. For example, a method that opens an Internet connection might take a user name, password, filename, protocol, and so on, but supply defaults if some of this information is not provided.

In this situation, it would be convenient to pass only the arguments to which the defaults did not apply. Another example is the **printf( )** method that is part of Java’s I/O library. As you will see in Chapter 20, it takes a variable number of arguments, which it formats and then outputs.

Prior to JDK 5, variable-length arguments could be handled two ways, neither of which was particularly pleasing. First, if the maximum number of arguments was small and known, then you could create overloaded versions of the method, one for each way the method could be called. Although this works and is suitable for some cases, it applies to only a narrow class of situations.

In cases where the maximum number of potential arguments was larger, or unknowable, a second approach was used in which the arguments were put into an array, and then the array was passed to the method. This approach is illustrated by the following program:

// Use an array to pass a variable number of

// arguments to a method. This is the old-style

// approach to variable-length arguments.

class PassArray {

static void vaTest(int v[]) {

System.out.print("Number of args: " + v.length +" Contents: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

// Notice how an array must be created to

// hold the arguments.

int n1[] = { 10 };

int n2[] = { 1, 2, 3 };

int n3[] = { };

vaTest(n1); // 1 arg

vaTest(n2); // 3 args

vaTest(n3); // no args

}

}

The output from the program is shown here:

Number of args: 1 Contents: 10

Number of args: 3 Contents: 1 2 3

Number of args: 0 Contents:

In the program, the method **vaTest( )** is passed its arguments through the array **v**. This old-style approach to variable-length arguments does enable **vaTest( )** to take an arbitrary number of arguments. However, it requires that these arguments be manually packaged into an array prior to calling **vaTest( )**. Not only is it tedious to construct an array each time **vaTest( )** is called, it is potentially error-prone. The varargs feature offers a simpler, better option.

A variable-length argument is specified by three periods (**…**). For example, here is how **vaTest( )** is written using a vararg:

static void vaTest(int ... v) {

This syntax tells the compiler that **vaTest( )** can be called with zero or more arguments. As a result, **v** is implicitly declared as an array of type **int[ ]**. Thus, inside **vaTest( )**, **v** is accessed using the normal array syntax. Here is the preceding program rewritten using a vararg:

// Demonstrate variable-length arguments.

class VarArgs {

// vaTest() now uses a vararg.

static void vaTest(int ... v) {

System.out.print("Number of args: " + v.length + " Contents: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

// Notice how vaTest() can be called with a

// variable number of arguments.

vaTest(10); // 1 arg

vaTest(1, 2, 3); // 3 args

vaTest(); // no args

}

}

The output from the program is the same as the original version.

There are two important things to notice about this program. First, as explained, inside **vaTest( )**, **v** is operated on as an array. This is because **v** *is* an array. The … syntax simply tells the compiler that a variable number of arguments will be used, and that these arguments will be stored in the array referred to by **v**. Second, in **main( )**, **vaTest( )** is called with different numbers of arguments, including no arguments at all. The arguments are automatically put in an array and passed to **v**. In the case of no arguments, the length of the array is zero.

A method can have “normal” parameters along with a variable-length parameter. However, the variable-length parameter must be the last parameter declared by the method. For example, this method declaration is perfectly acceptable:

int doIt(int a, int b, double c, int ... vals) {

In this case, the first three arguments used in a call to **doIt( )** are matched to the first three parameters. Then, any remaining arguments are assumed to belong to **vals**.

Remember, the varargs parameter must be last. For example, the following declaration

is incorrect:

int doIt(int a, int b, double c, int ... vals, boolean stopFlag) { // Error!

Here, there is an attempt to declare a regular parameter after the varargs parameter, which

is illegal.

There is one more restriction to be aware of: there must be only one varargs parameter.

For example, this declaration is also invalid:

int doIt(int a, int b, double c, int ... vals, double ... morevals) { // Error!

The attempt to declare the second varargs parameter is illegal.

Here is a reworked version of the **vaTest( )** method that takes a regular argument and a

variable-length argument:

// Use varargs with standard arguments.

class VarArgs2 {

// Here, msg is a normal parameter and v is a

// varargs parameter.

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

System.out.print(msg + v.length +

" Contents: ");

for(int x : v)

System.out.print(x + " ");

System.out.println();

}

public static void main(String args[])

{

vaTest("One vararg: ", 10);

vaTest("Three varargs: ", 1, 2, 3);

vaTest("No varargs: ");

}

}

The output from this program is shown here:

One vararg: 1 Contents: 10

Three varargs: 3 Contents: 1 2 3

No varargs: 0 Contents:

---------------------------------------------------------------------------------------------------------------