Module: Core Java

Session 4: Class Design and Encapsulation

* Method Overloading
* Overloading Constructors
* Using Objects as Parameters
* Passing Arguments
* Returning Objects
* A Closer Look at Recursion
* Importance of Access Control
* Understanding Static
* Understanding Final
* Nested and Inner Classes
* Command-line Arguments

**Objective**

At the end of this chapter, you will be able to:

* Know the function of Method Overloading process
* Overloading constructors
* Objects as parameters and pass arguments
* Find the way objects are returned
* Learn from example of recursion
* Work on the importance of access control
* Use static and final in a statement
* Prepare nested and inner classes
* Command-line arguments

**Method Overloading**

Method overloading is an important feature in Java, which allows the overloading of two or more methods within the same class.

Creation of several functions with the same name but different signature is called method overloading.

The methods have different parameter declarations. The process of overloading is central to statically typed programming languages that implement type checking in function calls. Method overloading feature not only re-uses method names, it also permits the execution of several vital functions.

For example, both the square function of ‘X’ (can be integer) and its return value (can be floating point number) can have overloading.

From the instrumental point of view, this feature should never be confused with ad-hoc polymorphism though overloading is one of the ways through which Java implements polymorphism. Java applies the “one interface, multiple methods” paradigm through method overloading. For those languages, where method overloading is not supported, each method must be given a unique name. However, frequently you might want to employ the same method for different types of data.

While invoking an overload method ask yourself why should overload methods differ in the type and/or number of their parameters? This is done because type and/or number of arguments are used in JAVA to determine which version of the overloaded method actually needs to be called. Overloaded methods might have different return types. However, these return types are not enough in themselves to distinguish between two versions of a method.

In JAVA, you can encounter a call to an overload method. However, remember that while doing so, JAVA only executes that version of the method whose parameters match the arguments in call.

**Note**: If in a program, multiple methods are invoked in order to perform diverse tasks, overloading method should be avoided.

**OverloadTestDemo.java** illustrates method overloading:

// This example will illustrate method overloading.

class OverloadTestDemo

{

void OVtest()

{

System.out.println("No parameters");

}

// Overloading OVtest for one integer parameter.

void OVtest(int a)

{

System.out.println("a: " + a);

}

// Overloading OVtest for two integer parameters.

void OVtest(int a, int b)

{

System.out.println("a and b: " + a + " " + b);

}

// Overloading OVtest for a double parameter

double OVtest(double a)

{

System.out.println("double a: " + a);

return a\*a;

}

}

class OverloadCallDemo

{

public static void main(String args[])

{

OverloadTestDemo ob = new OverloadTestDemo();

double result;

// call all versions of OVtest()

ob.OVtest();

ob.OVtest(20);

ob.OVtest(10, 30);

result = ob.OVtest(100.25);

System.out.println("Result of ob.OVtest(100.25): " + result);

}

}

**OverloadTestDemo.java** generates the following output:

No parameters

a: 20

a and b: 10 30

double a: 100.25

Result of ob.OVtest(100.25): 10050.0625

In OverloadTestDemo.java, **OVtest()** is overloaded four times. For the first method, no parameters are defined. The second one takes one integer parameter, the third takes two integer parameters and the fourth takes one **double** parameter.

Whenever an overloaded method is called during the course of a program, JAVA searches for a match between the arguments used to call the method and the method’s parameters. It is not necessary for the method to be exact all the time. JAVA’s automatic type conversion can play a role in overhead resolution under certain situations.

For example, consider **OverloadTestDemo1.java**:

// Automatic type conversions applied to overloading.

class OverloadTestDemo1

{

void OVtest()

{

System.out.println("No parameters");

}

// Overloading test for two integer parameters.

void OVtest(int a, int b)

{

System.out.println("a and b: " + a + " " + b);

}

// overloading test for a double parameter

void OVtest(double a)

{

System.out.println("Inside OVtest(double) a: " + a);

}

}

class OverloadCallDemo1

{

public static void main(String args[])

{

OverloadTestDemo1 ob = new OverloadTestDemo1();

int i = 78;

ob.OVtest();

ob.OVtest(5, 7);

ob.OVtest(i); // this will invoke OVtest(double)

ob.OVtest(1.3); // this will invoke OVtest(double)

}

}

**OverloadTestDemo1.java** generates the following output:

No parameters

a and b: 5 7

Inside OVtest(double) a: 78.0

Inside OVtest(double) a: 1.3

The **OVtest(int)** is not defined by **OverloadTestDemo1.java.** So, no matching method will be found whenever **OVtest()** is called with an integer argument inside **Overload.** What is the way out then? An integer can be automatically translated into a double in JAVA and this conversion can be used to resolve the call.

**Note:** Automatic type conversion in JAVA is employed only if no exact match is found.

“C” language doesn’t support method-overloading feature. Each and every function in “C” should have different name, even if they execute the same task. In Java, each absolute value method can use the same name. Infact, Java’s standard class library includes an absolute value method known as **abs()**, which is overloaded by Java’s math class to handle all numeric types. On the basis of the argument, Java compiler can identify and define which version of **abs()** to call.

In a nutshell, method overloading has certain explicit uses:

1. Method overloading allows related methods to be assessed by the use of a common name.
2. Overloading helps you to manage and monitor the complexity of the programs.

## Overloading Constructors

In the same way as several methods are employed and adjusted in a single class, you can load two or more constructors in a single class. Real-time programming actually calls for the induction of different constructors during the designing of programs. To be more precise, for most of the real-world classes that you create, overloaded constructors are a norm rather than an exception. Let’s refer to the following fragment code representing class **Cons1**.

//using constructors

class Cons1

{

private int x;

private int y;

Cons1() //default constructor

{

x = 10;

y = 20;

}

Cons1(int a, int b) //parameterized constructor

{

x = a;

y = b;

}

void calc()

{

int res = x + y;

System.out.println("Result is : "+res);

}

} //class over

class OverConsDemo1

{

public static void main(String args[])

{

Cons1 ob1 = new Cons1();

ob1. calc();

Cons1 ob2 = new Cons1(30,40);

ob2.calc();

}

}

In the code, it is quite manifest that first **Cons1()** requires no parameter and the other **Cons1()** constructor requires two parameters. This further implies that all declarations of **Cons1** objects must pass either no or two arguments to the **Cons1()** constructor. Since **Cons1()** requires two arguments, it’s an error to call it without them.

Following is another example implementing Constructor Overloading and trying to pass dimension for a Cuboid. Find out the volume of the same. **OverConsDemo2.java** contains an improved version of **OverConsDemo1.java**.

/\* Cuboid1 defines three constructors to initialize the dimensions of a box. \*/

class Cuboid1 {

double width;

double height;

double depth;

// constructor used when all dimensions are specified

Cuboid1(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions are specified

Cuboid1() {

width = -1; // use -1 to indicate

height = -1; // an uninitialized

depth = -1; // box

}

// constructor used when cube is created

Cuboid1(double len) {

width = height = depth = len;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

class OverConsDemo2 {

public static void main(String args[]) {

// creation of boxes using various constructors

Cuboid1 mycube1 = new Cuboid1(15, 50, 5);

Cuboid1 mycube2 = new Cuboid1();

Cuboid1 mycube3 = new Cuboid1(9);

double vol;

// deriving the volume of the first box

vol = mycube1.volume();

System.out.println("Volume of mybox1 is " + vol);

// deriving the volume of the second box

vol = mycube2.volume();

System.out.println("Volume of mybox2 is " + vol);

// deriving the volume of the cube

vol = mycube3.volume();

System.out.println("Volume of mycube3 is " + vol);

}

}

The output produced by **OverConsDemo2.java** is:

Volume of mycube1 is 3750.0

Volume of mycube2 is -1.0

Volume of mycube3 is 729.0

**Note**: The required overloaded constructor is called based upon the parameters specified when **new** is executed.

**Using Objects as Parameters**

Till now we have learnt how to overload methods and constructors. Now we should learn how to add objects to methods. For example, consider **PassObjectDemo.java**:

// passing of objects to methods.

class ObjTest {

int a, b;

ObjTest(int i, int j) {

a = i;

b = j;

}

// return false if o is equal to the invoking object

boolean equals(ObjTest o) {

if(o.a == a && o.b == b) return false;

else return true;

}}

class PassObjectDemo {

public static void main(String args[]) {

ObjTest ob1 = new ObjTest(333, 33);

ObjTest ob2 = new ObjTest(333, 33);

ObjTest ob3 = new ObjTest(1, 1);

System.out.println("ob1 == ob2: " + ob1.equals(ob2));

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

}

}

**PassObjectDemo.java** generates the following output:

ob1 == ob2: false

ob1 == ob3: true

**Note**: **equals()** method incorporated in **ObjTest** is employed to measure the equality or non-equality of two objects. Parameter ‘o’ in **equals()** identifies **ObjTest** as its type.

Quite often, the usage of object parameters involves the application of constructors and you may want to create a new object, which is objectively same as some of the existing object. In order to execute an application like this, all you need to do is define a constructor that takes an object of its class as a parameter. In **OverConsDemo3.java**, the **Cuboid** version allows one object to initialize another:

// Box allows one object to initialize another object.

class Cuboid2 {

double width;

double height;

double depth;

// constructs a clone of an object

Cuboid2(Cuboid2 ob) { // pass object to constructor

width = ob.width;

height = ob.height;

depth = ob.depth;

}

// constructor used when all dimensions are specified

Cuboid2(double w, double h, double d) {

width = w;

height = h;

depth = d;

}

// constructor used when no dimensions are specified

Cuboid2() {

width = 1; // use -1 to indicate

height = 1; // an uninitialized

depth = 1; // box

}

// constructor used when a cube is created

Cuboid2(double len) {

width = height = depth = len;

}

// compute and return volume

double volume() {

return width \* height \* depth;

}

}

class OverConsDemo3 {

public static void main(String args[]) {

// creation of boxes using various constructors

Cuboid2 mycube1 = new Cuboid2(70, 5, 100);

Cuboid2 mycube2 = new Cuboid2();

Cuboid2 mycube3 = new Cuboid2(5);

Cuboid2 myclone = new Cuboid2(mycube1);

double vol;

// deriving the volume of first box

vol = mycube1.volume();

System.out.println("Volume of mycube1 is " + vol);

// deriving the volume of second box

vol = mycube2.volume();

System.out.println("Volume of mycube2 is " + vol);

// deriving the volume of cube

vol = mycube3.volume();

System.out.println("Volume of mycube3 is " + vol);

// deriving the volume of clone

vol = myclone.volume();

System.out.println("Volume of clone is " + vol);

}

}

**OverConsDemo3.java** generates the following output:

Volume of mycube1 is 35000.0

Volume of mycube2 is 1.0

Volume of mycube3 is 125.0

Volume of clone is 35000.0

#### Passing Arguments

One important thing that you can do is passing an argument into a subroutine. There are two ways by which you can do it. The first way is called as ‘call-by-value’ method and the second one is termed as the ‘call-by-reference’ method. The first technique copies the value of an argument into the parameter of a subroutine whereas following the second method, an argument reference can be passed to the parameter. For the first method, changes to the parameter of the subroutine have no formal effect on the argument. For the second method, the reference is used to access the actual argument specified in the call which further implies that changes initiated to the parameter affects the argument that is used to call the subroutine.

**Note**: Java uses both these methods, depending upon what is passed.

When you pass a simple type to a method, it is passed by value. The parameter that accepts the argument has no effect outside the method. Please refer to **CallByValueDemo.java** to understand how simple types are passed by value.

// Simple types are passed by value.

class TestDemo1 {

void meth(int i, int j) {

i \*= 2;

j /= 2;

}}

class CallByValueDemo {

public static void main(String args[]) {

TestDemo1 ob = new TestDemo1();

int a = 28, b = 29;

System.out.println("a and b before call: " + a + " " + b);

ob.meth(a, b);

System.out.println("a and b at the end of the call: " + a + " " + b);

}

}

The output of **CallByValueDemo.java** is shown here:

a and b before call: 28 29

a and b at the end of the call: 28 29

The operations taking place inside **meth()** does not have any effect on the values of a and bthat are used in the call. Their values did not change.

This situation, however, changes radically when an object is passed to a method. Why? Because, objects are passed by reference. Remember that when a variable of a class type is created, you are actually creating only a reference to an object. Hence, when a reference is passed to this method, the parameter that receives it will refer only to the same object as that referred to by the argument. Objects are, therefore, passed to methods by use of call-by-reference. The object used as an argument is affected by the changes to the object inside the method.

Consider the **CallByRefDemo.java** as an example:

// Objects are passed by reference.

class TestDemo2 {

int x, y;

TestDemo2(int i, int j) {

x = i;

y = j;

}

// pass an object

void meth(Test o) {

o.x \*= 2;

o.y /= 2;

}

}

class CallByRefDemo {

public static void main(String args[]) {

TestDemo2 ob = new TestDemo2(7, 8);

System.out.println("ob.x and ob.y before call: " + ob.x + " " + ob.y);

ob.meth(ob);

System.out.println("ob.x and ob.y after call: " + ob.x + " " + ob.y);

}

}

**CallByRefDemo.java** generates the following output:

ob.x and ob.y before call: 7 8

ob.x and ob.y after call: 14 4

The object used as an argument has been affected by the actions inside **meth()**. An interesting point is that when an object reference is passed to a method, a call-by-value is used to pass by the reference itself. The value that is being passed refers to an object. The copy of that value will also refer to the same object that its corresponding argument does.

The call-by-value is used to pass a simple type to a method. You will need to use the call-by-reference to pass the objects.

#### Returning Objects

Returning of objects is one of the marked features in java language. A method can return any form of data, including class types. In **RetObjectDemo.java**, the **incrementByTen()** method returns an object in which the value of **x** is seven greater than it is in the invoking object.

// Returning an object.

class TestDemo3 {

int x;

TestDemo3(int y) {

x = y;

}

TestDemo3 incrementByTen() {

TestDemo3 temp = new TestDemo3(x+7);

return temp;

}

}

class RetObjectDemo {

public static void main(String args[]) {

TestDemo3 ob1 = new TestDemo3(2);

Testdemo3 ob2;

ob2 = ob1.incrementByTen();

System.out.println("ob1.x: " + ob1.x);

System.out.println("ob2.x: " + ob2.x);

ob2 = ob2.incrementByTen();

System.out.println("ob2.x after second increase: " + ob2.x);

}

}

The output generated by **RetObjectDemo.java** is shown here:

ob1.x: 2

ob2.x: 9

ob2.x after second increase: 16

**Note**: A new object is created and a reference to it is returned whenever you call incrementByTen(). The new object created is dynamically allocated to the memory. After returning the object, the method terminates and the created object continues to exist as long as there is a reference to it somewhere in the program. What will happen if the object goes un-referenced? Of course, the object will acquire garbage status and will be reclaimed by the garbage collector when it returns next time.

**A Closer Look at Recursion**

Noted linguist and political theorist Noam Chomsky was one of the pioneers to use recursion mechanism in languages. He embedded one or more sentences in sentences. Recursion forms an intrinsic part of computer programming. In Java Programming language, recursion permits a method to call itself. Thus a recursive method is considered to be one that can call itself.

Introduction of an ‘if’statement (to force the method to return without the recursive call being executed) is mandatory while writing recursive methods. If ‘if’statement is not implemented once you call the method, it will never return. The programmers are strongly recommended to use println()statements liberally during the invocation and execution of recursive methods so that they can watch what is going on and abort execution if you see that you have made a mistake.

A classic example of recursion is the computation of the factorial of a number. The factorial of a number *X* is the product of all the whole numbers between 1 and *X.* For example, 6 factorial is 1 × 2 × 3 × 4 × 5 × 6, or 720. Here is how a factorial can be computed by use of a recursive method:

// A simple example of recursion:

class FactorialDemo {

// this is a recursive function

int fact(int x) {

int result;

if(x==1)

return 1;

result = fact(x-1) \* x;

return result;

}

}

class RecursionDemo1 {

public static void main(String args[]) {

FactorialDemo f = new FactorialDemo();

System.out.println("Factorial of 6 is " + f.fact(6));

System.out.println("Factorial of 7 is " + f.fact(7));

System.out.println("Factorial of 9 is " + f.fact(9));

}

}

**RecursionDemo1.java** generates the following output:

Factorial of 6 is 720

Factorial of 7 is 5040

Factorial of 9 is 362880

**Note**: In the above example, when **fact()** is called with an argument of 1, the function returns 1; otherwise it returns the product of **fact(x–1)\*x**. To evaluate this expression, **fact()** is called with **x–1**. This process repeats until **x** equals 1 and the calls to the method begin returning.

New local variables and parameters are allocated memory storage on the stack, whenever a method calls itself. The method code is then executed with these new variables from the start. It is important to note here that a recursive call does not make a new copy of the method and it is only the arguments that are new. With the return of each recursive call, the old local variables and parameters are removed from the stack, and the recursive operation resumes at the point of the call inside the method.

Recursive functions have certain invincible advantages. One of the main advantages of using recursion is that recursive methods can be used to create simpler versions of several algorithms.

A computer programmer might face certain pertinent problems while using recursion. Because of the added function calls, recursive versions may operate slower in relation to the iterative equivalent. There is the possibility of a stack overrun. There is also a likelihood of stack collapse because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables. In the event of a stock collapse, the Java run-time system will cause an exception.

In order to understand recursion in a more comprehensive way, let us consider another example. The recursive method printArray() prints the first y elements in the array values.

// Another example that uses recursion.

class RecursiveTestDemo {

int values[];

RecursiveTestDemo(int y) {

values = new int[y];

}

// display array -- recursively

void printArray(int y) {

if(y==0) return;

else printArray(y-1);

System.out.println("[" + (y-1) + "] " + values[y-1]);

}

}

class RecursionDemo2 {

public static void main(String args[]) {

RecursiveTestDemo ob = new RecursiveTestDemo(10);

int y;

for(y=0; y<10; y++)

ob.values[y] = y;

ob.printArray(10);

}

}

**RecursionDemo2.java** generates the following output:

[0] 0

[1] 1

[2] 2

[3] 3

[4] 4

[5] 5

[6] 6

[7] 7

[8] 8

[9] 9

#### Importance of Access Control

Java, in order to retain the compactness of data, has set definite protocols and levels to manage and access data. Java defines a default access level. Access control is a very important feature in Java language. Access control is another form of data encapsulation employing which you can prevent data misuse and data tampering. To control the access of data as well as to restrict access to the data members of a class you have certain defined methods. In real-time classes, you allow operations on data only through methods. When these methods are judicially employed and implemented, a class creates a “black box” which may be used. It is important to note here that the inner ingredients of the class are not open to tampering. Certain major and important aspects of access control are principally related to inheritance or packages.

In access control, access specifier plays a pivotal role. Java provides for three different kinds of specifiers viz. **public**, **private**, and **protected**. The access specifier not only tells you how a member can be accessed, it also tells you how to create and modify the classes’ declaration statement. An access specifier precedes the rest of a member’s type specification. Please refer to the following program code to understand access control specifiers:

public int i;

private double j;

There are moments where the specifier is not categorized. In that event, the member of a class comes (by default) under ‘**public**’ within its own package. It is needless to say here that for most of the real-time programs all members of a class use the default ‘**public**’.

**Note**: Member of an unspecified public class cannot be accessed by members of other packages.

Whenever a member of a class is modified with the aid of a public specifier, then that member can be accessed by any other code. In real-time programming, it is rightly justified to allow an instance variable to be public. If a member of a class is specified as private, then that member can only be accessed by other members of the same class. This is the reason why main()has always been preceded by the publicspecifier. Specifier ‘protected’applies only when inheritance is involved.

To comprehend the effects of public and private access, consider AccessTestDemo1.java:

/\* This program demonstrates the difference between public and private.

\*/

class TestDemo3 {

int x; // default access

public int y; // public access

private int z; // private access, methods to access c

void setz(int i) { // set z's value

z = i;

}

int getz() { // get z's value

return z;

}

}

class AccessTestDemo1 {

public static void main(String args[]) {

TestDemo3 ob = new TestDemo3();

// These are OK, x and y may be accessed directly

ob.x = 17;

ob.y = 18; // This is not OK and will cause an error

ob.z = 19; // Error! You must access z through its methods

ob.setz(19); // OK

System.out.println("x, y, and z: " + ob.x + " " + ob.y + " " + ob.getz());

}

}

**AccessTestDemo1.java** generates the following output:

x, y, and z: 17 18 19

In **AccessTestDemo1.java**, we see that inside the TestDemoclass, **x** uses default access (equivalent to accessing the public class), **y** is overtly specified as **public** and member **z** is given private access. Member **z** can not be accessed by code outside of its class and hence inside **AccessTest1** class, **z** cannot be used directly. It must be accessed through its public methods: **setz()** and **getz()**.

**TestStackDemo1.java** demonstrates the improved **Stack** class.

import java.io.\*;

import java.util.\*;

class TestStackDemo1 {

public static void main(String args[]) {

Stack mystack1 = new Stack();

Stack mystack2 = new Stack();

// push some numbers onto the stack

for(int i=0; i<5; i++) mystack1.push(i);

for(int i=5; i<10; i++) mystack2.push(i);

// pop those numbers off the stack

System.out.println("Stack in mystack1:");

for(int i=0; i<5; i++)

System.out.println(mystack1.pop());

System.out.println("Stack in mystack2:");

for(int i=0; i<5; i++)

System.out.println(mystack2.pop());

// these statements are not legal

// mystack1.tos = -2;

// mystack2.stck[3] = 100;

}

}

**TestStackDemo1.java** generates the following output:

Stack in mystack1:

4

3

2

1

0

Stack in mystack2:

9

8

7

6

5

With regard to the above example, though it is true that the push() and pop() methods provide for a controlled interface to the stack, yet it is possible for another part of the program to bypass these methods and access the stack directly.

## Understanding Static

It is very important to know how the keyword ‘static’ functions in advanced Java applications. Under standard conditions, a class member must be accessed in combination with an object of its class. But it is sometimes feasible to create a member that can be used by itself, without reference to a specific instance. In order to initialize this sort of a class member, the declaration should be preceded by the keyword ‘static’. A class member, declared as ‘static’, can be accessed without reference to any object and before any object/objects of its class is/are created. Both methods as well as variables can be defined as static. ‘main()’ is the most common and universal example of a static member and has to be invoked before the initialization of objects.

**Note:** In essence, the instance variables declared as static are global variables. No copy of a static variable is initiated whenever objects of its class are defined. The same static variable is shared by all instances of the class.

It is important for you to understand that there are certain limitations that are central to methods that are declared as ‘static’. These are:

1. The methods are capable of calling only other static methods.
2. Only static data can be accessed by the methods.

**Note**: In order to initialize your static variables you may need to do computation because the static block that you declare gets executed exactly once, when the class is loaded for the first time.

**StaticTestDemo.java** speaks about a class that includes a static method, some static variables and a static initialization block.

// Demonstrate static variables, methods, and blocks.

class StaticTestDemo {

static int x = 3;

static int y = 4;

static int z;

static void statmeth(int x) {

System.out.println("x = " + x);

System.out.println("y = " + y);

System.out.println("z = " + z);

}

//Starting of static block

static {

System.out.println("Static block initialized.");

z = y \* 8;

}

public static void main(String args[]) {

statmeth(31);

}

}

**StaticTestDemo.java** generates the following output**:**

Static block initialized

x = 31

y = 4

z = 32

As soon as the **StaticTestDemo** class is loaded, all of the static statements are run. First, x is set to 3, and y set to 4 and z set to 0. After this, the static block executes (printing a message) and finally, z is initialized to y \* 8 or 32. Then main() is called, which calls statmeth(), passing 31 to x. The three println() statements refer to the two static variables x, y and z.

**Note**: It is illegal to refer to any instance variables inside of a static method.

‘static’ methods and variables can be used independently of any object, outside the class in which these methods and variables are defined. In order to use ‘**static**’ methods and variables independently, all you need to do is stipulate the name of their class followed by the dot operator.

In the event that you wish to call a static method from outside its class, you can do so by using the following fragment code:

classname.method()

In the above fragment code, *classname* is the name of the class within which the static method is declared. In a similar fashion, a static variable can be accessed. Using a dot operator on the name of the class, you can initiate a variable.

Please refer to **StaticByNameDemo.java**

class StaticTestDemo {

static int j = 22;

static int k = 31;

static void callme() {

System.out.println("j = " + j);

}

}

class StaticByNameDemo {

public static void main(String args[]) {

StaticTestDemo.callme();

System.out.println("k = " + StaticTestDemo.j);

System.out.println("k = " + StaticTestDemo.k);

}

}

**StaticByNameDemo.java** generates the following output:

j = 22

k = 31

**Note**: In **StaticByNameDemo.java,** inside main(), static method callme() and static variable j & k are accessed outside of their class.

Understanding Final

To know classes and methods in a much more simpler way, it is very important to recognize how we can manipulate variables. In fact to be more precise, we can assert the forms of a variable. We can declare a variable as ‘**final’**. In the ‘**final’** form of a variable you can’t initiate changes in contents and final variable is essentially a constant. Those variables that are declared as final do not occupy memory on a per-instance basis. The omnipresence of the keyword **‘final’** also stretches to methods and we can even declare a method to be **‘final’**. Please refer to the below-mentioned examples:

final int FILE\_NEW = 1;

final int FILE\_OPEN = 2;

final int FILE\_SAVE = 3;

final int FILE\_SAVEAS = 4;

final int FILE\_QUIT = 5;

**Note**: If the above fragment codes are applied in real-time java programming, you will see that parts of your program can use FILE\_SAVE, FILE\_QUIT etc. (as if they were constants). Please also note that choosing all uppercase identifiers for final variables is a common coding practice.

#### Nested and Inner class

As we all know that Java is identified in terms of classes and their functional attributes. Classes in Java exist in different forms allowing class members to accommodate themselves in different formats and positions. A class represents object and embodies all the essential features of a particular set of objects. In real-life terms, you don’t define objects. You rather define classes of objects.

Apart from the general form of existence, classes exist in the form of ‘Nested class’ and ‘Inner class’. Both ‘Nested and Inner classes’ are explained in details in Fig 1.



Fig 1: Classification of classes

###### Nested classes

As you can create sub-systems inside systems and sub-sets inside sets, similarly you can create and initialize sub-classes within classes. Java language authorizes you to define a class within another class. Please refer to the below-mentioned fragment code to identify and understand a ‘nested class’.

class OuterClass {

...

class NestedClass {

...

}

}

One of the primary attributes of a nested class is that it has access to other members of the enclosing class, even if they are declared private. Nested classes can be declared private, public, protected, or package private.

**Note**: It is important to note here that outer classes can only be declared public or package private.

Nested classes are categorized into two types static and non-static. Those nested classes that are declared as static are called static nested classes. Non-static nested classes are called inner classes. Refer to the following fragment code for further details.

class OuterClass {

...

static class StaticNestedClass {

...

}

class InnerClass {

...

}

}

###### Reasons for using nested Classes

There are certain advatages of using nested classes:

* We can logically and coherently group and demarcate classes.
* With nested classes, we get higher amounts of data encapsulation. You can well hide or reveal data of a nested class in concordance with the program requirements.
* Nested classes facilitate the process of coding and thus enhance data manipulation.
* Nested classes initiate and expedite referencing of multiple objects.
* With nested classes you can demarcate between global, local and not so local classes.

###### Static Nested Classes

Static nested classes are defined by their inherent attributes and characteristics. Though a static nested class is associated with its outer class yet it cannot refer directly to instance variables or methods defined in its enclosing class. Static nested classes can use instance variables or methods only through an object reference. At an instrumental level, a static nested class is a top-level class that is cased in another top-level class for expediency.

The following class name is used to access static nested classes.

OuterClass.StaticNestedClass

To generate an object for the static nested class the following fragment code is used:

OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();

###### Non-static Classes or Inner Classes

Inner classes contribute substantially towards application programming in Java. Inner classes are associated with instances of their enclosing classes and have direct access to object's methods and fields. As inner classes are associated with instances, they cannot define any static members themselves. It is noteworthy to mention here that objects that are instances of an inner class exist within an instance of the outer class. Please refer to the following codes for more details:

class OuterClass {

...

class InnerClass {

...

}

}

In order to instantiate an inner class, you first have to initialize the outer class. Only after that will you be able to create the inner object within the outer object. The fragment code for the same would look like:

OuterClass.InnerClass innerObject = outerObject.new InnerClass();

Please refer to the following Figure to understand how inner and outer classes coexist.

**OUTER CLASSES**

**INNER CLASSES**

Fig 2: An InnerClass Exists Within an Instance of OuterClass

**Note**: Instance of InnerClass has direct access to the methods and fields of its enclosing instance.

### Local and Anonymous Inner Classes

Inner classes are present in two different forms. These are: local inner classes and anonymous inner classes. A local inner class is declared within the body of a method. When the same local inner class exists within the body of a method without a definite class name, the same is regarded as anonymous inner classes. You will encounter such classes in advanced Java programming.

#### Command-line Arguments

During intensive programming, times come when you need to infuse (during program execution) additional information or input into an existing program. This is done with the aid of adding command-line arguments to main(). Command-line argument is specific information that follows the program’s name on the command-line when executed. Command-line arguments are stored as strings in the String array that is passed to main(). During program execution, you have to physically convert numeric values to their internal forms.

Refer to the following fragment code to know how command-line arguments are infused into a program.

// Display all command-line arguments.

class CommandLineDemo {

public static void main(String args[]) {

for(int i=0; i<args.length; i++)

System.out.println("args[" + i + "]: " + args[i]);

}

}

**Summary**

In this chapter you have learnt one of the main pillars of object oriented programming i.e., Polymorphism. You have learnt how same name can be given to different methods, which are functionally doing the same task. Overloading helps you to manage and monitor the complexity of the programs.

* Now you know that methods can be given same name, differing only in parameters being passed to them.
* Even constructs can be overloaded.
* When a simple type is passed to a method, it is done by use of call-by-value. Objects are passed by use of call-by-reference.
* Recursion is a method calling itself. One of the main advantages of using recursion is that recursive methods can be used to create simpler versions of several algorithms.
* Java provides three different access specifies: private, protected, public.
* A class member, which is declared as ‘**static**’, can be accessed without reference to any object and before any object/objects of its class is/are created.
* In order to make a variable constant, we use the access specifies **final.**
* Java language authorizes you to define a class within another class.
* **Nested classes** are categorized into two types **static** and **non-static**.
* Command-line arguments are stored as strings in the String array that is passed to main().