A Closer Look at 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.

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

//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

However, this match need not always be exact. In some cases, Java’s automatic type conversions can play a role in overload. 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.

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

**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 to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine. As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed. When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.

**Java Pass By Reference And Pass By Value**

There are basically two types of techniques for passing the parameters in Java. The first one is pass-by-value and the second one is pass-by-reference. One thing to remember here is that when a [**primitive typ**](https://www.softwaretestinghelp.com/java/java-data-types-loops-arrays-switch-assertions/)e is passed to a method, then it is done by the use of pass-by-value.

However, all the non-primitive types that include objects of any class are always implicitly passed by use of pass-by-reference. Basically, pass-by-value means that the actual value of the variable is passed and pass-by-reference means the memory location is passed where the value of the variable is stored.

In this example, we will showcase how to pass a parameter by using pass-by-value which is also known as call-by-value. Here we have initialized a variable ‘a’ with some value and used the pass-by-value technique to show how the value of the variable remains unchanged. In the next segment, we will try to show a similar example, but we will use non-primitives.

**public** **class** Example {

    /\*

     \*  The original value of a will remain unchanged in

     \*  case of call-by-value

     \*/

**int** a = 10;

**void** call(**int** a) {

        // this local variable a is subject to change in its value

        a = a+10;

    }

**public** **static** **void** main(String[] args) {

        Example eg = **new** Example();

        System.out.println("Before call-by-value: " + eg.a);

        /\*

         \* Passing an integer 50510 to the call() method. The value of

         \* 'a' will still be unchanged since the passing parameter is a

         \* primitive type.

         \*/

        eg.call(50510);

        System.out.println("After call-by-value: " + eg.a);

    }

}

Output:

Before call-by-value: 10

After call-by-value: 10

Java Passing Object: Pass by Reference Example

In this example, we will see how to pass any object of a class using pass-by-reference.

As you can see, when we have passed the object reference as a value instead of a value, the original value of the variable ‘a’ is changed to 20. This is because of the changes in the called method.

|  |
| --- |
| **public** **class** Example {      /\*       \*  The original value of 'a' will be changed as we are trying       \*  to pass the objects. Objects are passed by reference.       \*/  **int** a = 10;  **void** call(Example eg) {          eg.a = eg.a+10;      }  **public** **static** **void** main(String[] args) {          Example eg = **new** Example();          System.out.println("Before call-by-reference: " + eg.a);          // passing the object as a value using pass-by-reference          eg.call(eg);          System.out.println("After call-by-reference: " + eg.a);      }  } |

**Output:**

Before call-by-reference: 10

After call-by-reference: 10

**Recursion**

Java supports recursion. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be recursive. The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3 ×, or 6. Here is how a factorial can be computed by use of a recursive method:

//A simple example of recursion.

**class** Factorial {

// this is a recursive method

**int** fact(**int** n) {

**int** result;

**if** (n == 1)

**return** 1;

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

**return** result;

}

}

**class** Recursion {

**public** **static** **void** main(String args[]) {

Factorial f = **new** Factorial();

System.***out***.println("Factorial of 3 is " + f.fact(3));

System.***out***.println("Factorial of 4 is " + f.fact(4));

System.***out***.println("Factorial of 5 is " + f.fact(5));

}

}

The output from this program is shown here:

Factorial of 3 is 6

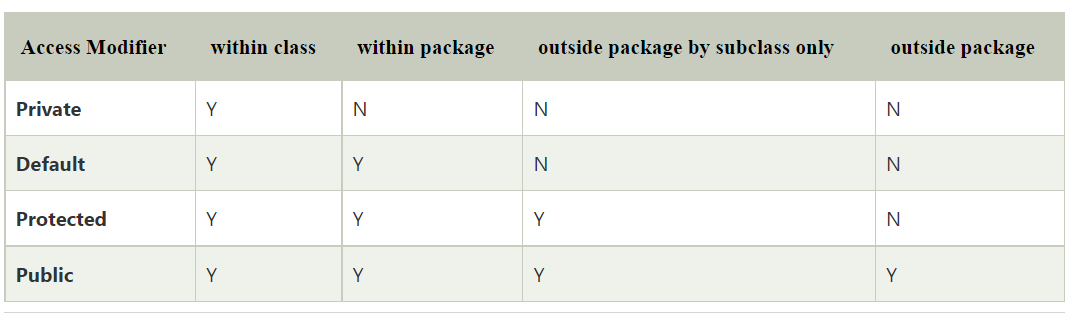
Factorial of 4 is 24

Factorial of 5 is 120

**Access Modifiers in Java**

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:

1. Default – No keyword required
2. Private
3. Protected
4. Public



1. **Default**: When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

//Java program to illustrate default modifier

**package** p1;

//Class Geeks is having Default access modifier

**class** Geek

{

**void** display()

{

System.out.println("Hello World!");

}

}

//Java program to illustrate error while

//using class from different package with

//default modifier

**package** p2;

**import** p1.\*;

//This class is having default access modifier

**class** GeekNew

{

**public** **static** **void** main(String args[])

{

// Accessing class Geek from package p1

Geeks obj = **new** Geek();

obj.display();

}

}

**Output:**

Compile time error

**NOTE**: 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.

1. **Private**:  The private access modifier is specified using the keyword **private**.

The methods or data members declared as private are accessible only **within the class** in which they are declared. Any other **class of**the **same package will not be able to access** these members.

Top-level classes or interfaces cannot be declared as private because

* + 1. private means “only visible within the enclosing class”.
    2. protected means “only visible within the enclosing class and any subclasses”

Hence these modifiers in terms of application to classes, apply only to nested classes and not on top-level classes

In this example, we will create two classes A and B within the same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

//Java program to illustrate error while

//using class from different package with

//private modifier

**package** p1;

**class** A

{

**private** **void** display()

{

System.out.println("GeeksforGeeks");

}

}

**class** B

{

**public** **static** **void** main(String args[])

{

A obj = **new** A();

// Trying to access private method

// of another class

obj.display();

}

}

**Output:**

error: display() has private access in A

obj.display();

1. **Protected**: The protected access modifier is specified using the keyword **protected.**

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

//Java program to illustrate

//protected modifier

**package** p1;

//Class A

**public** **class** A

{

**protected** **void** display()

{

System.out.println("GeeksforGeeks");

}

}

//Java program to illustrate

//protected modifier

**package** p2;

**import** p1.\*; // importing all classes in package p1

//Class B is subclass of A

**class** B **extends** A

{

**public** **static** **void** main(String args[])

{

B obj = **new** B();

obj.display();

}

}

**Output:**

GeeksforGeeks

1. **Public**: The public access modifier is specified using the keyword **public.**

The public access modifier has the **widest scope** among all other access modifiers.

Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

**Understanding Static:**

There will be times when you will want to define a class member that will be used independently of any object of that class. 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. 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:

1. They can only directly call other static methods.
2. They can only directly access static data.
3. They cannot refer to this or super in any way. (The keyword super relates to inheritance and is described in the next chapter.)

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.

//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()

**Introducing final:**

A field can be declared as **final**. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor. The first approach is the most common. Here is an example:

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;

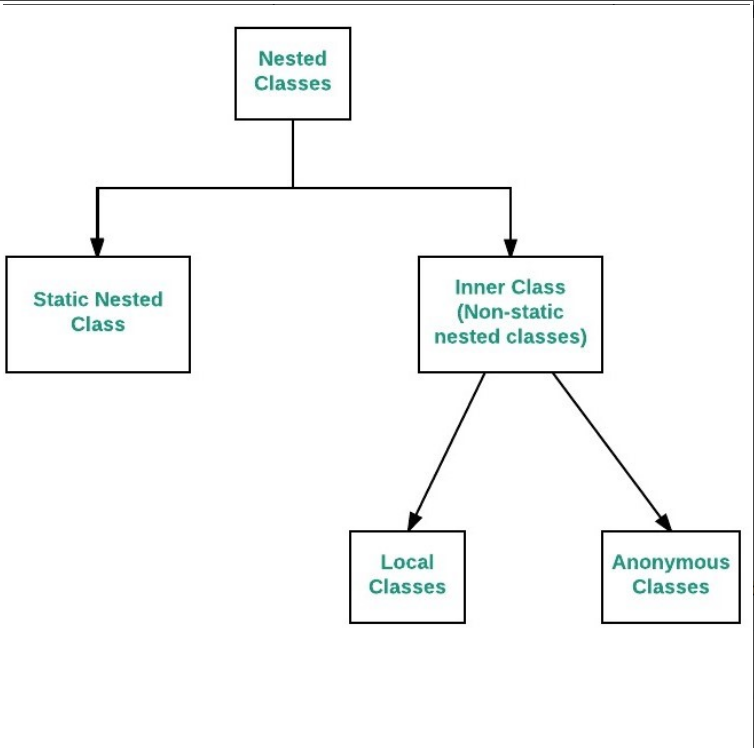
Subsequent parts of your program can now use FILE\_OPEN, etc., as if they were constants, without fear that a value has been changed. It is a common coding convention to choose all uppercase identifiers for final fields, as this example shows. In addition to fields, both method parameters and local variables can be declared final. Declaring a parameter final prevents it from being changed within the method. Declaring a local variable final prevents it from being assigned a value more than once.

**Nested Class in JAVA**

In Java, it is possible to define a class within another class, such classes are known as nested classes. They enable you to logically group classes that are only used in one place, thus this increases the use of encapsulation, and creates more readable and maintainable code.

1. The scope of a nested class is bounded by the scope of its enclosing class. Thus in below example, class *NestedClass* does not exist independently of class *OuterClass*.
2. A nested class has access to the members, including private members, of the class in which it is nested. The reverse is also true i.e., the enclosing class can access the members of the nested class.
3. A nested class is also a member of its enclosing class.
4. As a member of its enclosing class, a nested class can be declared *private*, *public*, *protected*, or *package private*(default).
5. Nested classes are divided into two categories:
   * **static nested class:** Nested classes that are declared *static* are called static nested classes.
   * **inner class:**An inner class is a non-static nested class.

Static Nested Class: In the case of static nested class, Without an outer class object existing, there may be a static nested class object. i.e., an object of a static nested class is not strongly associated with the outer class object.



**Advantage of Java inner classes**

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

**Types of Nested classes**

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class

1. **Local inner class:**

A class i.e., created inside a method, is **called local inner class in java**. Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. Therefore, it would be better if you include it within other classes.

**public** **class** localInner1 {

**private** **int** data = 30;// instance variable

**void** display() {

**// inner local class**

**class** Local {

**void** msg() {

System.***out***.println(data);

}

}

Local l = **new** Local();

l.msg();

}

**public** **static** **void** main(String args[]) {

localInner1 obj = **new** localInner1();

obj.display();

}

}

Output: 30

1. **Anonymous inner class:**

It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overriding methods of a class or interface, without having to actually subclass a class.

**Syntax:**

// Test can be interface,abstract/concrete class

Test t = new Test()

{

// data members and methods

public void test\_method()

{

........

........

}

};

Now let us do discuss the difference between regular class(normal classes) and Anonymous Inner class

* A normal class can implement any number of interfaces but the anonymous inner class can implement only one interface at a time.
* A regular class can extend a class and implement any number of interfaces simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.
* For regular/normal class, we can write any number of constructors but we can’t write any constructor for anonymous Inner class because the anonymous class does not have any name and while defining constructor class name and constructor name must be same.

Anonymous inner classes are generic created via below listed two ways as follows:

1. Class (may be abstract or concrete)
2. Interface

Types of Anonymous Inner Class

Based on declaration and behavior, there are 3 types of anonymous Inner classes:

1. Anonymous Inner class that extends a class
2. Anonymous Inner class that implements an interface
3. Anonymous Inner class that defines inside method/constructor argument

**Type 1: Anonymous Inner class that extends a class**

We can have an anonymous inner class that extends a class. For example, we know that we can create a thread by extending a Thread class. Suppose we need an immediate thread but we don’t want to create a class that extends Thread class all the time. With the help of this type of Anonymous Inner class, we can define a ready thread.

//Java program to illustrate creating an immediate thread

//Using Anonymous Inner class that extends a Class

//Main class

**class** MyThread {

// Main driver method

**public** **static** **void** main(String[] args) {

// Using Anonymous Inner class that extends a class

// Here a Thread class

Thread t = **new** Thread() {

// run() method for the thread

**public** **void** run() {

// Print statement for child thread

// execution

System.***out***.println("Child Thread");

}

};

// Starting the thread

t.start();

// Displaying main thread only for readability

System.***out***.println("Main Thread");

}

}

**Output**

Main Thread

Child Thread

**Type 2: Anonymous Inner class that implements an interface**

We can also have an anonymous inner class that implements an interface. For example, we also know that by implementing Runnable interface we can create a Thread. Here we use an anonymous Inner class that implements an interface.

**Example**

//Java program to illustrate defining a thread

//Using Anonymous Inner class that implements an interface

//Main class

**class** MyThread {

// Main driver method

**public** **static** **void** main(String[] args) {

// Here we are using Anonymous Inner class

// that implements a interface i.e. Here Runnable

// interface

Runnable r = **new** Runnable() {

// run() method for the thread

**public** **void** run() {

// Print statement when run() is invoked

System.***out***.println("Child Thread");

}

};

// Creating thread in main() using Thread class

Thread t = **new** Thread(r);

// Starting the thread using start() method

// which invokes run() method automatically

t.start();

// Print statement only

System.***out***.println("Main Thread");

}

}

**Type 3:**Anonymous Inner class that defines inside method/constructor argument

Anonymous inner classes in method/constructor arguments are often used in graphical user interface (GUI) applications. To get you familiar with syntax lets have a look at the following program that creates a thread using this type of Anonymous Inner class

**Example**

//Java program to illustrate defining a thread

//Using Anonymous Inner class that define inside argument

//Main class

**class** MyThread {

// Main driver method

**public** **static** **void** main(String[] args)

{

// Using Anonymous Inner class that define inside

// argument

// Here constructor argument

Thread t = **new** Thread(**new** Runnable() {

**public** **void** run()

{

System.***out***.println("Child Thread");

}

});

t.start();

System.***out***.println("Main Thread");

}

}