# Java Archive

## Setting up the environment in Java

Java is a general-purpose computer programming language that is concurrent, class-based, object-oriented etc.
Java applications are typically compiled to **bytecode** that can run on any Java virtual machine (JVM) regardless of computer architecture.The latest version is **Java 8**.

Below are the environment settings for both Linux and Windows. JVM, JRE and JDK  all three are platform dependent because configuration of each Operating System is different. But, Java is platform independent.

There are few things which must be clear before setting up the environment

1. **JDK**(Java Development Kit) : JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.
2. **JRE**(Java Runtime Environment) : JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.
3. **JVM:** JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms.

**Steps for Setting up Java Environment for Windows**

1. Java8 JDK is available at Download Java 8.
   For Windows(32 bit),you have to click second last link and for Windows(64 bit),you have to click last link as illustrated below.
   Capture
2. After download, run the *.exe* file and follow the instructions to install Java on your machine. Once you installed Java on your machine, you have to setup environment variable.
3. Go to **Control Panel -> System and Security -> System.**
   Click on Advanced System Setting option.You will see as illustrated below.
   Now click on **Environment Variables** as highlighted in image.
   Capture
4. Now click on **New tab** under user variable.
   Give the **variable name** of your choice (example:JAVA_HOME)
   For **variable value** click on Browse Directory tab as illustrated and Copy the path of bin folder of Java.
   Capture
5. Now under Browse Directory go to **PATH = C:\Program Files\Java\jdk\bin** and click OK.
   By default java is installed in Program Files or Program Files(x86) folder. In case, you have installed java at any other location,then select that path.
   Capture
6. Save the settings and you are done !! Now to check whether installation is done correctly, open command prompt and type *java -version*. You will see that java is running on your machine.

**Steps for Linux**

In linux, there are several ways to install java. But we will refer to simplest and easy way to install java using terminal. For linux we will install OpenJDK. OpenJDK is a free and open source implementation of the Java programming language.

1. Go to **Application -> Accessories -> Terminal**.
2. Type command as below..

```
sudo apt-get install openjdk-8-jdk
```

3. For “JAVA_HOME” (Environment Variable) type command as shown below, in “Terminal” using your installation path…(Note: the default path is as shown, but if you have install OpenJDK at other location then set that path.)

```
export JAVA_HOME = /usr/lib/jvm/java-8-openjdk
```

4. For “PATH” (Environment Value) type command as shown below, in “Terminal” using your installation path…Note: the default path is as shown, but if you have install OpenJDK at other location then set that path.)

```
export PATH = $PATH:/usr/lib/jvm/java-8-openjdk/bin
```

5. You are done !! Now to check whether installation is done correctly, type *java -version* in the Terminal.You will see that java is running on your machine.

**Popular Java Editors/IDE** :

- **Notepad/gedit** : They are simple text-editor for writing java programs. Notepad is available on Windows and gedit is available on Linux.
- **Eclipse IDE** : It is most widely used IDE(Integrated Development Environment) for developing softwares in java. You can download Eclipse from here.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java
TechTips
Java-Basics

# Beginning Java programming with Hello World Example

The process of Java programming can be simplified in three steps:

- Create the program by typing it into a text editor and saving it to a file – HelloWorld.java.
- Compile it by typing “javac HelloWorld.java” in the terminal window.
- Execute (or run) it by typing “java HelloWorld” in the terminal window.

Below given  program is the simplest program of Java printing “Hello World” to the screen. Let us try to understand every bit of code step by step.

```
/* This is a simple Java program.
   FileName : "HelloWorld.java". */
class HelloWorld
{
    // Your program begins with a call to main().
    // Prints "Hello, World" to the terminal window.
    public static void main(String args[])
    {
        System.out.println("Hello World");
    }
}
```

Output:

```
Hello World
```

The “Hello World!” program consists of three primary components: the HelloWorld class definition, the main method and source code comments. Following explanation will provide you with a basic understanding of the code:

1. **Class definition:**This line uses the keyword **class** to declare that a new class is being defined.

   ```
   class HelloWorld
   ```

   **HelloWorld** is an identifier that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace **{** and the closing curly brace **}** .

2. **main method:** In Java programming language, every application must contain a main method whose signature is:

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

   public: So that JVM can execute the method from anywhere.
   static: Main method is to be called without object.
   The modifiers public and static can be written in either order.
   void: The main method doesn't return anything.
   main(): Name configured in the JVM.
   String[]: The main method accepts a single argument:
             an array of elements of type String.
   ```

   Like in C/C++, main method is the entry point for your application and will subsequently invoke all the other methods required by your program.

3. The next line of code is shown here. Notice that it occurs inside main( ).

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

   This line outputs the string “Hello World” followed by a new line on the screen. Output is actually accomplished by the built-in println( *)* method. **System** is a predefined class that provides access to the system, and **out** is the variable of type output stream that is connected to the console.

4. Comments: They can either be multi-line or single line comments.

   ```
   /* This is a simple Java program.
   Call this file "HelloWorld.java". */
   ```

   This is a multiline comment. This type of comment must begin with /* and end with */. For single line you may directly use // as in C/C++.

**Important Points** :

- The name of the class defined by the program is HelloWorld, which is same as name of file(HelloWorld.java). This is not a coincidence. In Java, all codes must reside inside a class and there is at most one public class which contain main() method.
- By convention, the name of the main class(class which contain main method) should match the name of the file that holds the program.

**Compiling the program** :

- After successfully setting up the environment, we can open terminal in both Windows/Unix and can go to directory where the file – HelloWorld.java is present.
- Now, to compile the HelloWorld program, execute the compiler – javac , specifying the name of the **source** file on the command line, as shown:

  ```
  javac HelloWorld.java
  ```

- The compiler creates a file called HelloWorld.class (in present working directory) that contains the bytecode version of the program. Now, to execute our program, **JVM**(Java Virtual Machine) needs to be called using java, specifying the name of the **class** file on the command line, as shown:

  ```
  java HelloWorld
  ```

  This will print “Hello, World” to the terminal screen.

**In Windows**

Capture

**In Linux**

VirtualBox_Hadoop_ubuntu_SN_07_02_2017_03_37_06

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Basics

---

# Java Naming Conventions

Below are some naming conventions of for java programming language. They must be followed while developing software in java for good maintenance and readability of code. Java uses CamelCase as a practice for writing names of methods, variables, classes, packages and constants.

**Camel case in Java Programming :** It consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital.

1. **Classes and Interfaces** :
   - Class names should be **nouns**, in mixed case with the **first** letter of each internal word capitalized. Interfaces name should also be capitalized just like class names.
   - Use whole words and must avoid acronyms and abbreviations.

   Examples:

```
Interface  Bicycle
Class MountainBike implements Bicyle

Interface Sport
Class Football implements Sport
```

2. **Methods :**
   - Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalized.

   Examples:

```
void changeGear(int newValue);
void speedUp(int increment);
void applyBrakes(int decrement);
```

3. **Variables :** Variable names should be short yet meaningful.
   - Should **not** start with underscore('_') or dollar sign '$' characters.
   - Should be mnemonic i.e, designed to indicate to the casual observer the intent of its use.
   - **One-character variable names should be avoided** except for temporary variables.
   - Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

   Examples:

```
  // variables for MountainBike class
  int speed = 0;
  int gear = 1;
```

4. **Constant variables:**
   - Should be **all uppercase** with words separated by underscores ("_").
   - There are various constants used in predefined classes like Float, Long, String etc.

   Examples:

```
static final int MIN_WIDTH = 4;

// Some  Constant variables used in predefined Float class
```

```
public static final float POSITIVE_INFINITY = 1.0f / 0.0f;
public static final float NEGATIVE_INFINITY = -1.0f / 0.0f;
public static final float NaN = 0.0f / 0.0f;
```

5. **Packages:**
   - The prefix of a unique package name is always written in **all-lowercase ASCII letters** and should be one of the top-level domain names, like com, edu, gov, mil, net, org.
   - Subsequent components of the package name vary according to an organization's own internal naming conventions.

   Examples:

```
com.sun.eng
com.apple.quicktime.v2

// java.lang packet in JDK
java.lang
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

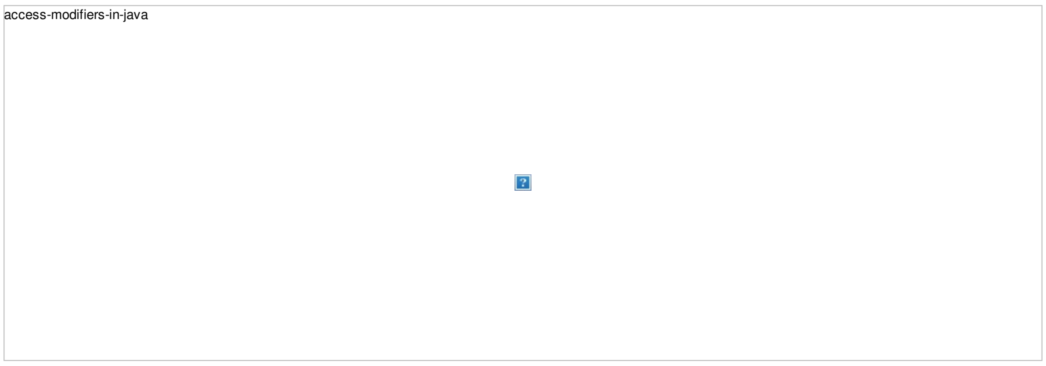
## GATE CS Corner    Company Wise Coding Practice

Java

# 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**.

   In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of second 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
```

2. **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 same package will not be able to access** these members.
   - Classes or interface can not be declared as private.

   In this example, we will create two classes A and B within 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();
```

3. **protected**: The protected access modifier is specified using the keyword **protected**.
   - The methods or data members declared as protected are **accessible within same package or sub classes in different package.**

   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
```

4. **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 which are declared as public are **accessible from every where** in the program. There is no restriction on the scope of a public data members.

```
//Java program to illustrate
//public modifier
package p1;
public class A
{
   public void display()
      {
          System.out.println("GeeksforGeeks");
      }
}
package p2;
import p1.*;
class B
{
    public static void main(String args[])
      {
          A obj = new A;
          obj.display();
      }
}
```

Output:

```
GeeksforGeeks
```

**Important Points:**

- If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
- Avoid public fields except for constants.

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Classes and Objects in Java

Classes and Objects are basic concepts of Object Oriented Programming which revolve around the real life entities.

**Class**

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers** : A class can be public or private or protected or has default access.
2. **Class name:** The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. **Body:** The class body surrounded by braces, { }.

Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

There are various types of classes that are used in real time applications such as nested classes, anonymous classes, lambda expressions.

**Object**

It is a basic unit of Object Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consist of :

1. **State** : It is represented by attributes of an object. It also reflect the properties of an object.
2. **Behavior** : It is represented by methods of an object. It also reflects the response of an object with other objects.
3. **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Example of an object : dog

Blank Diagram - Page 1 (5)

Objects correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".

**Declaring Objects (Also called instantiating a class)**

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.

Example :

Blank Diagram - Page 1 (3)

As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variable, type must be strictly a concrete class name. In general,we **can't** create objects of an abstract class or an interface.

```
Dog tuffy;
```

If we declare reference variable(tuffy) like this, its value will be undetermined(null) until an object is actually created and assigned to it. Simply declaring a reference variable does not create an object.

**Initializing an object**

The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.

```
// Class Declaration

public class Dog
{
    // Class Variables
    String name;
    String breed;
    int age;
    String color;

    // Constructor Declaration of Class
    public Dog(String name, String breed,
                   int age, String color)
    {
        this.name = name;
        this.breed = breed;
        this.age = age;
        this.color = color;
    }

    // method 1
    public String getName()
    {
        return name;
    }

    // method 2
    public String getBreed()
    {
        return breed;
    }

    // method 3
    public int getAge()
    {
        return age;
    }

    // method 4
    public String getColor()
    {
        return color;
    }

    @Override
    public String toString()
    {
        return("Hi my name is "+ this.getName()+
               ".\nMy breed,age and color are " +
               this.getBreed()+"," + this.getAge()+
               ","+ this.getColor());
    }

    public static void main(String[] args)
    {
        Dog tuffy = new Dog("tuffy","papillon", 5, "white");
        System.out.println(tuffy.toString());
    }
}
```

Output:

```
Hi my name is tuffy.
My breed,age and color are papillon,5,white
```

- This class contains a single constructor. We can recognize a constructor because its declaration uses the same name as the class and it has no return type. The Java compiler differentiates the constructors based on the number and the type of the arguments. The constructor in the *Dog* class takes four arguments. The following statement provides "tuffy","papillon",5,"white" as values for those arguments:

  ```
  Dog tuffy = new Dog("tuffy","papillon",5, "white");
  ```

  The result of executing this statement can be illustrated as :

  Untitled

**Note :** All classes have at least **one** constructor. If a class does not explicitly declare any, the Java compiler automatically provides a no-argument constructor, also called the default constructor. This default constructor calls the class parent's no-argument constructor (as it contain only one statement i.e super();), or the *Object* class constructor if the class has no other parent (as Object class is parent of all classes either directly or indirectly).

**Ways to create object of a class**

There are four ways to create objects in java.Strictly speaking there is only one way(by using *new* keyword),and the rest internally use *new* keyword.

- **Using new keyword** : It is the most common and general way to create object in java. Example:

  ```

  // creating object of class Test
  Test t = new Test();
  ```

- **Using Class.forName(String className) method** : There is a pre-defined class in java.lang package with name Class. The forName(String className) method returns the Class object associated with the class with the given string name.We have to give the fully qualified name for a class. On calling new Instance() method on this Class object returns new instance of the class with the given string name.

  ```
  // creating object of public class Test
  // consider class Test present in com.p1 package
  Test obj = (Test)Class.forName("com.p1.Test").newInstance();
  ```

- **Using clone() method**: clone() method is present in Object class. It creates and returns a copy of the object.

```
// creating object of class Test
Test t1 = new Test();

// creating clone of above object
Test t2 = (Test)t1.clone();
```

- **Deserialization** : De-serialization is technique of reading an object from the saved state in a file. Refer Serialization/De-Serialization in java

```
FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Object obj = in.readObject();
```

**Creating multiple objects by one type only (A good practice)**

- In real-time, we need different objects of a class in different methods. Creating a number of references for storing them is not a good practice and therefore we declare a static reference variable and use it whenever required. In this case,wastage of memory is less. The objects that are not referenced anymore will be destroyed by Garbage Collector of java. Example:

```
Test test = new Test();
test = new Test();
```

- In inheritance system, we use parent class reference variable to store a sub-class object. In this case, we can switch into different subclass objects using same referenced variable. Example:

```
class Animal {}

class Dog extends Animal {}
class Cat extends Animal {}

public class Test
{
   // using Dog object
   Animal obj = new Dog();

   // using Cat object
   obj = new Cat();
}
```

**Anonymous objects**

Anonymous objects are the objects that are instantiated but are not stored in a reference variable.

- They are used for immediate method calling.
- They will be destroyed after method calling.
- They are widely used in different libraries. For example, in AWT libraries, they are used to perform some action on capturing an event(eg a key press).
- In example below, when a key is button(referred by the btn) is pressed, we are simply creating anonymous object of EventHandler class for just calling handle method.

```
btn.setOnAction(new EventHandler()
{
   public void handle(ActionEvent event)
   {
      System.out.println("Hello World!");
   }
});
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Basics

# How JVM Works – JVM Architecture?

JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Run Environment).

Java applications are called WORA (Write Once Run Everywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, a *.class* file(contains byte-code) with the same filename is generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

jvm

Image Source : https://en.wikipedia.org/wiki/Java_virtual_machine

**Class Loader Subsystem**

It is mainly responsible for three activities.

- Loading

- Linking
- Initialization

**Loading :** The Class loader reads the *.class* file, generate the corresponding binary data and save it in method area. For each *.class* file, JVM stores following information in method area.

- Fully qualified name of the loaded class and its immediate parent class.
- Whether *.class* file is related to Class or Interface or Enum
- Modifier, Variables and Method information etc.

After loading *.class* file, JVM creates an object of type Class to represent this file in the heap memory. Please note that this object is of type Class predefined in *java.lang* package. This Class object can be used by the programmer for getting class level information like name of class, parent name, methods and variable information etc. To get this object reference we can use *getClass()* method of Object class.

```
// A Java program to demonstrate working of a Class type
// object created by JVM to represent .class file in
// memory.
import java.lang.reflect.Field;
import java.lang.reflect.Method;

// Java code to demonstrate use of Class object
// created by JVM
public class Test
{
    public static void main(String[] args)
    {
        Student s1 = new Student();

        // Getting hold of Class object created
        // by JVM.
        Class c1 = s1.getClass();

        // Printing type of object using c1.
        System.out.println(c1.getName());

        // getting all methods in an array
        Method m[] = c1.getDeclaredMethods();
        for (Method method : m)
            System.out.println(method.getName());

        // getting all fields in an array
        Field f[] = c1.getDeclaredFields();
        for (Field field : f)
            System.out.println(field.getName());
    }
}

// A sample class whose information is fetched above using
// its Class object.
class Student
{
    private String name;
    private int roll_No;

    public String getName()  {  return name;   }
    public void setName(String name) { this.name = name; }
    public int getRoll_no()  { return roll_No;  }
    public void setRoll_no(int roll_no) {
        this.roll_No = roll_no;
    }
}
```

Output:

```
Student
getName
setName
getRoll_no
setRoll_no
name
roll_No
```

**Note :** For every loaded *.class* file, only **one** object of Class is created.

```
Student s2 = new Student();
// c2 will point to same object where
// c1 is pointing
Class c2 = s2.getClass();
System.out.println(c1==c2); // true
```

**Linking :** Performs verification, preparation, and (optionally) resolution.

- *Verification* : It ensures the correctness of *.class* file i.e. it check whether this file is properly formatted and generated by valid compiler or not. If verification fails, we get run-time exception *java.lang.VerifyError.*
- *Preparation* : JVM allocates memory for class variables and initializing the memory to default values.
- *Resolution* : It is the process of replacing symbolic references from the type with direct references. It is done by searching into method area to locate the referenced entity.

**Initialization :** In this phase, all static variables are assigned with their values defined in the code and static block(if any). This is executed executed from top to bottom in a class and from parent to child in class hierarchy.
In general there are three class loaders :

- *Bootstrap class loader* : Every JVM implementation must have a bootstrap class loader, capable of loading trusted classes. It loads core java API classes present in *JAVA_HOME/jre/lib* directory. This path is popularly known as bootstrap path. It is implemented in native languages like C, C++.
- *Extension class loader* : It is child of bootstrap class loader. It loads the classes present in the extensions directories *JAVA_HOME/jre/lib/ext*(Extension path) or any other directory specified by the java.ext.dirs system property. It is implemented in java by the *sun.misc.Launcher$ExtClassLoader* class.
- *System/Application class loader* : It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$ExtClassLoader* class.

```
// Java code to demonstrate Class Loader subsystem
public class Test
{
    public static void main(String[] args)
    {
        // String class is loaded by bootstrap loader, and
        // bootstrap loader is not Java object, hence null
        System.out.println(String.class.getClassLoader());

        // Test class is loaded by Application loader
        System.out.println(Test.class.getClassLoader());
    }
}
```

Output:

```
null
sun.misc.Launcher$AppClassLoader@73d16e93
```

**Note :** JVM follow Delegation-Hierarchy principle to load classes. System class loader delegate load request to extension class loader and extension class loader delegate request to boot-strap class loader. If class found in boot-strap path, class is loaded otherwise request again transfers to extension class loader and then to system class loader. At last if system class loader fails to load class, then we get run-time exception *java.lang.ClassNotFoundException*.

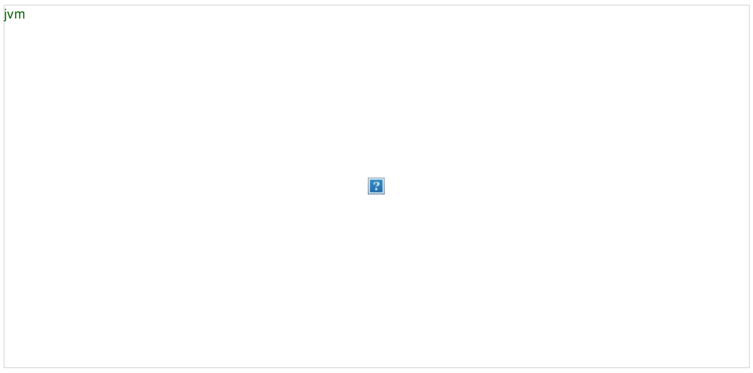


Image Source: http://javarevisited.blogspot.in/2012/12/how-classloader-works-in-java.html

**JVM Memory**

**Method area :**In method area, all class level information like class name, immediate parent class name, methods and variables information etc. are stored, including static variables. There is only one method area per JVM, and it is a shared resource.
**Heap area :**Information of all objects is stored in heap area. There is also one Heap Area per JVM. It is also a shared resource.

**Stack area :**For every thread, JVM create one run-time stack which is stored here. Every block of this stack is called activation record/stack frame which store methods calls. All local variables of that method are stored in their corresponding frame. After a thread terminate, it's run-time stack will be destroyed by JVM. It is not a shared resource.

**PC Registers :**Store address of current execution instruction of a thread. Obviously each thread has separate PC Registers.

**Native method stacks :**For every thread, separate native stack is created. It stores native method information.

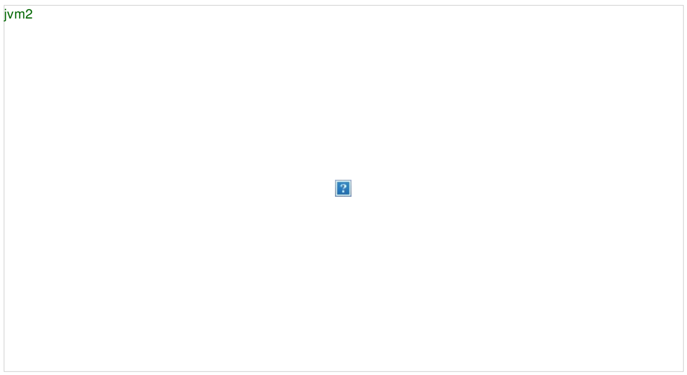


Image Source : http://java.scjp.jobs4times.com/fund/fund2.png

**Execution Engine**

Execution engine execute the *.class* (bytecode). It reads the byte-code line by line, use data and information present in various memory area and execute instructions. It can be classified in three parts :-

- *Interpreter* : It interprets the bytecode line by line and then executes. The disadvantage here is that when one method is called multiple times, every time interpretation is required.
- *Just-In-Time Compiler(JIT)* : It is used to increase efficiency of interpreter.It compiles the entire bytecode and changes it to native code so whenever interpreter see repeated method calls,JIT provide direct native code for that part so re-interpretation is not required,thus efficiency is improved.
- *Garbage Collector* : It destroy un-referenced objects.For more on Garbage Collector,refer Garbage Collector.

**Java Native Interface (JNI) :**

It is a interface which interacts with the Native Method Libraries and provides the native libraries(C, C++) required for the execution. It enables JVM to call C/C++ libraries and to be called by C/C++ libraries which may be specific to hardware.

**Native Method Libraries :**

It is a collection of the Native Libraries(C, C++) which are required by the Execution Engine.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Packages In Java

**Package** in Java is a mechanism to encapsulate a group of classes, sub packages and interfaces. Packages are used for:

- Preventing naming conflicts. For example there can be two classes with name Employee in two packages, college.staff.cse.Employee and college.staff.ee.Employee
- Making searching/locating and usage of classes, interfaces, enumerations and annotations easier
- Providing controlled access: protected and default have package level access control. A protected member is accessible by classes in the same package and its subclasses. A default member (without any access specifier) is accessible by classes in the same package only.
- Packages can be considered as data encapsulation (or data-hiding).

All we need to do is put related classes into packages. After that we can simply write a import a class from existing packages and use it in our program. A packages is container of group of related classes where some of the classes are accessible are exposed and others are kept for internal purpose.
We can reuse existing classes from the packages as many time as we need it in our program.

**How packages work?**

Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.
**Package naming conventions :** Packages are named in reverse order of domain names, i.e., org.geeksforgeeks.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.
**Adding a class to a Package :** We can add more classes to an created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.

**Subpackages:** Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.
**Example :**

```
import java.util.*;
```

**util** is a subpackage created inside **java** package.

**Accessing classes inside a package**

Consider following two statements :

```
// import the Vector class from util package.
import java.util.vector;

// import all the classes from util package
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;

// Only mentioned class of this package will be accessible.
import package.classname;

// Class name is generally used when two packages have same
// class name. For example in below code both packages have
// date class so using fully qualified name to avoid conflict
import java.util.Date;
import my.packag.Date;
```

```
// Java program to demonstrate accessing of members when
// a corresponding classes are imported and not imported.
import java.util.Vector;

public class ImportDemo
{
   public ImportDemo()
   {
      // java.util.Vector is imported, hence we are
      // able to access directly in our code.
      Vector newVector = new Vector();

      // java.util.ArrayList is not imported, hence
      // we were referring to it using the complete
      // package.
      java.util.ArrayList newList = new java.util.ArrayList();
   }

   public static void main(String arg[])
   {
      new ImportDemo();
   }
}
```

**Types of packages:**

packages

**Built-in Packages**
These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:
1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) **java.io:** Contains classed for supporting input / output operations.
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) **java.applet:** Contains classes for creating Applets.
5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
6) **java.net:** Contain classes for supporting networking operations.

**User-defined packages**
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement

being the **package names**.

```
// Name of the package must be same as the directory
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "GeeksforGeeks";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();

      obj.getNames(name);
   }
}
```

**Note : MyClass.java** must be saved inside the **myPackage** directory since it is a part of the package.

**Using Static Import**

Static import is a feature introduced in **Java** programming language ( versions 5 and above ) that allows members ( fields and methods ) defined in a class as public **static** to be used in Java code without specifying the class in which the field is defined.
Following program demonstrates **static import** :

```
// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo
{
   public static void main(String args[])
   {
        // We don't need to use 'System.out'
        // as imported using static.
        out.println("GeeksforGeeks");
   }
}
```

Output:

```
GeeksforGeeks
```

**Handling name conflicts**

The only time we need to pay attention to packages is is when we have a name conflict . For example both, java.util and java.sql packages have a class named Date. So if we import both packages in program as follows:

```
import java.util.*;
import java.sql.*;

//And then use Date class , then we will get a compile time error :

Date today ; //ERROR-- java.util.Date or java.sql.Date?
```

The compiler will not be able to figure out which Date class do we want. This problem can be solved by using specific import statement:

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

If we need both Date classes then , we need to use full package name every time we declare new object of that class.
For Example:

```
java.util.Date deadLine = new java.util.Date();
java.sql.Date today = new java.sql.Date();
```

**Directory structure**

The package name is closely associated with the directory structure used to store the classes. The classes (and other entities) belonging to a specific package are stored together in the same directory. Furthermore, they are stored in a sub-directory structure specified by its package name. For example, the class Circle of package com.zzz.project1.subproject2 is stored as "$BASE_DIR\com\zzz\project1\subproject2\Circle.class", where $BASE_DIR denotes the base directory of the package. Clearly, the "dot" in the package name corresponds to a sub-directory of the file system.

The base directory ($BASE_DIR) could be located anywhere in the file system. Hence, the Java compiler and runtime must be informed about the location of the $BASE_DIR so as to locate the classes. This is accomplished by an environment variable called CLASSPATH. CLASSPATH is similar to another environment variable PATH, which is used by the command shell to search for the executable programs.

**Setting CLASSPATH**:
CLASSPATH can be set by any of the following ways:

- CLASSPATH can be set permanently in the environment: In Windows, choose control panel ? System ? Advanced ? Environment Variables ? choose "System Variables" (for all the users) or "User Variables" (only the currently login user) ? choose "Edit" (if CLASSPATH already exists) or "New" ? Enter "CLASSPATH" as the variable name ? Enter the required directories and JAR files (separated by semicolons) as the value (e.g., ".;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar"). Take note that you need to include the current working directory (denoted by '.') in the CLASSPATH.
  To check the current setting of the CLASSPATH, issue the following command:
  ```
  > SET CLASSPATH
  ```
- CLASSPATH can be set temporarily for that particular CMD shell session by issuing the following command:
  ```
  > SET CLASSPATH=.;c:\javaproject\classes;d:\tomcat\lib\servlet-api.jar
  ```
- Instead of using the CLASSPATH environment variable, you can also use the command-line option -classpath or -cp of the javac and java commands, for example,

```
> java –classpath c:\javaproject\classes com.abc.project1.subproject2.MyClass3
```

**Illustration of user-defined packages:**
Creating our first package:
File name – ClassOne.java

```
package package_name;

public class ClassOne {
 public void methodClassOne() {
  System.out.println("Hello there its ClassOne");
 }
}
```

Creating our second package:
File name – ClassTwo.java

```
package package_one;

public class ClassTwo {
 public void methodClassTwo(){
  System.out.println("Hello there i am ClassTwo");
 }
}
```

Making use of both the created packages:
File name – Testing.java

```
import package_one.ClassTwo;
import package_name.ClassOne;

public class Testing {
 public static void main(String[] args){
  ClassTwo a = new ClassTwo();
  ClassOne b = new ClassOne();
  a.methodClassTwo();
  b.methodClassOne();
 }
}
```

Output:

```
Hello there i am ClassTwo
Hello there its ClassOne
```

Now having a look at the directory structure of both the packages and the testing class file:
Directory structure

**Important points:**

1. Every class is part of some package.
2. If no package is specified, the classes in the file goes into a special unnamed package (the same unnamed package for all files).
3. All classes/interfaces in a file are part of the same package. Multiple files can specify the same package name.
4. If package name is specified, the file must be in a subdirectory called name (i.e., the directory name must match the package name).
5. We can access public classes in another (named) package using: **package-name.class-name**

**Related Article:** Quiz on Packages in Java
**Reference:** http://pages.cs.wisc.edu/~hasti/cs368/JavaTutorial/NOTES/Packages.html
This article is contributed by **Nikhil Meherwal** and Prateek Agarwal. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

## Data types in Java

There are majorly two types of languages. First one is **Statically typed language** where each variable and expression type is already known at compile time.Once a variable is declared to be of a certain data type, it cannot hold values of other data types.Example: C,C++, Java. Other, **Dynamically typed languages:** These languages can receive different data types over the time. Ruby, Python

Java is **statically typed and also a strongly typed language** because in Java, each type of data (such as integer, character, hexadecimal, packed decimal, and so forth) is predefined as part of the programming language and all constants or variables defined for a given program must be described with one of the data types.

Java has two categories of data:

- Primitive data (e.g., number, character)
- Object data (programmer created types)

**Primitive data**

Primitive data are only single values; they have no special capabilities. There are 8 primitive data types

primitive data types in java

**boolean**

boolean data type represents only one bit of information **either true or false** . Values of type boolean are not converted implicitly or explicitly (with casts) to any other type. But the programmer can easily write conversion code.

```
// A Java program to demonstrate boolean data type
class GeeksforGeeks
{
    public static void main(String args[])
    {
        boolean b = true;
        if (b == true)
            System.out.println("Hi Geek");
    }
}
```

Output:

```
Hi Geek
```

**byte**

The byte data type is an 8-bit signed two's complement integer.The byte data type is useful for saving memory in large arrays.

- Size: 8-bit
- Value: -128 to 127

```
// Java program to demonstrate byte data type in Java
class GeeksforGeeks
{
    public static void main(String args[])
    {
        byte a = 126;

        // byte is 8 bit value
        System.out.println(a);

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

        // It overflows here because
        // byte can hold values from -128 to 127
        a++;
        System.out.println(a);

        // Looping back within the range
        a++;
        System.out.println(a);
    }
}
```

Output:

```
126
127
-128
-127
```

**short**

The short data type is a 16-bit signed two's complement integer. Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.

- **Size:** 16 bit
- **Value:** -32,768 to 32,767 (inclusive)

**int**

It is a 32-bit signed two's complement integer.

- **Size:** 32 bit
- **Value:** $-2^{31 \text{ to } 2^{31} -1}$

Note: In Java SE 8 and later, we can use the int data type to represent an unsigned 32-bit integer, which has value in range [0, $2^{32}$-1]. Use the Integer class to use int data type as an unsigned integer.

**long:**

The long data type is a 64-bit two's complement integer.

- Size: 64 bit
- Value: $-2^{63}$ to $2^{63}$-1.

Note: In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of $2^{64}$-1. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.

**Floating point Numbers : float and double**

**float**

The float data type is a single-precision 32-bit IEEE 754 floating point. Use a float (instead of double) if you need to save memory in large arrays of floating point numbers.

- **Size:** 32 bits
- **Suffix :** F/f Example: 9.8f

**double:**
The double data type is a double-precision 64-bit IEEE 754 floating point. For decimal values, this data type is generally the default choice.

Note: Both float and double data types were designed especially for scientific calculations, where approximation errors are acceptable. If accuracy is the most prior concern then, it is recommended not to use these data types and use BigDecimal class instead.
Please see this for details: Rounding off errors in Java
**char**
The char data type is a single 16-bit Unicode character. A char is a single character.

- Value: '\u0000' (or 0) to '\uffff' 65535

```
// Java program to demonstrate primitive data types in Java
class GeeksforGeeks
{
    public static void main(String args[])
    {
        // declaring character
         char a = 'G';

        // Integer data type is generally
        // used for numeric values
        int i=89;

        // use byte and short if memory is a constraint
        byte b = 4;

        // this will give error as number is
        // larger than byte range
        // byte b1 = 7888888955;

        short s = 56;

        // this will give error as number is
        // larger than short range
        // short s1 = 87878787878;


        // by default fraction value is double in java
        double d = 4.355453532;

        // for float use 'f' as suffix
        float f = 4.7333434f;

        System.out.println("char: " + a);
        System.out.println("integer: " + i);
        System.out.println("byte: " + b);
        System.out.println("short: " + s);
        System.out.println("float: " + f);
        System.out.println("double: " + d);
    }
}
```

Output:

```
char: G
integer: 89
byte: 4
short: 56
float: 4.7333436
double: 4.355453532
```

This article is contributed by **Shubham Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# What are C++ features missing in Java?

Following features of C++ are not there in Java.

No pointers
No sizeof operator
No scope resolution operator
Local variables in functions cannot be static
No Multiple Inheritance
No Operator Overloading
No preprocessor and macros
No user suggested inline functions
No goto
No default arguments
No unsigned int in Java
No -> operator in java
No stack allocated objects in java
No delete operator in java due to java's garbage collection
No destructor in java
No typedef in java
No global variables, no global function because java is pure OO.
No friend functions
No friend classes
No templates in java

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

## Does JVM create object of Main class (the class with main())?

Consider following program.

```
class Main {
    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Output:

```
Hello
```

Does JVM create an object of class Main?
The answer is "No". We have studied that the reason for main() static in Java is to make sure that the main() can be called without any instance. To justify the same, we can see that the following program compiles and runs fine.

```
// Not Main is abstract
abstract class Main {
    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Output:

```
Hello
```

Since we can't create object of abstract classes in Java, it is guaranteed that object of class with main() is not created by JVM.

This article is contributed by **Narendra Koli**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

## Myth about the file name and class name in Java

The first lecture note given during java class is "In java file name and class name should be the same". When the above law is violated a compiler error message will appear as below

```
/***** File name: Trial.java ******/
public class Geeks
{
   public static void main(String[] args) {
       System.out.println("Hello world");
   }
}
```

Output:

```
javac Trial.java
Trial.java:9: error: class Geeks is public, should be
                declared in a file named Geeks.java
public class Geeks
^
1 error
```

But the myth can be violated in such a way to compile the above file.

```
/***** File name: Trial.java ******/
class Geeks
{
    public static void main(String[] args) {
        System.out.println("Hello world");
    }
}
```

```
Step 1: javac Trial.java
```

Step1 will create a Geeks.class (byte code) without any error message since the class is not public.

```
Step 2: java Geeks
```

Now the output will be **Hello world**

The myth about the file name and class name should be same only when the class is declared in
**public**.

The above program works as follows :

javaex

Now this .class file can be executed. By the above features some more miracles can be done. It is possible to have many classes in a java file. For debugging purposes this approach can be used. Each class can be executed separately to test their functionalities(only on one condition: Inheritance concept should not be used).

But in general it is good to follow the myth.

For example:

```
/*** File name: Trial.java ***/
class ForGeeks
{
   public static void main(String[] args){
      System.out.println("For Geeks class");
   }
}

class GeeksTest
{
   public static void main(String[] args){
      System.out.println("Geeks Test class");
   }
}
```

When the above file is compiled as **javac Trial.java** will create two .class files as **ForGeeks.class and GeeksTest.class** .
Since each class has separate main() stub they can be tested individually.
When **java ForGeeks** is executed the output is **For Geeks class**.
When **java GeeksTest** is executed the output is **Geeks Test class**.

This article is contributed by **Sowmya.L.R**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# ‘this’ reference in Java

‘this’ is a reference variable that refers to the current object.

Following are the ways to use ‘this’ keyword in java :

**1. Using ‘this’ keyword to refer current class instance variables**

```
//Java code for using 'this' keyword to
//refer current class instance variables
class Test
{
    int a;
    int b;

    // Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
    }

    void display()
    {
        //Displaying value of variables a and b
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test(10, 20);
        object.display();
    }
}
```

Output:

```
a = 10  b = 20
```

**2. Using this() to invoke current class constructor**

```
// Java code for using this() to
// invoke current class constructor
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        this(10, 20);
        System.out.println("Inside  default constructor \n");
    }

    //Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println("Inside parameterized constructor");
    }

    public static void main(String[] args)
    {
        Test object = new Test();
    }
}
```

Output:

```
Inside parameterized constructor
Inside  default constructor
```

**3. Using ‘this’ keyword to return the current class instance**

```
//Java code for using 'this' keyword
//to return the current class instance
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that returns current class instance
    Test get()
    {
        return this;
    }

    //Displaying value of variables a and b
    void display()
    {
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get().display();
    }
}
```

Output:

```
a = 10  b = 20
```

**4. Using ‘this’ keyword as method parameter**

```
// Java code for using 'this'
// keyword as method parameter
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that receives 'this' keyword as parameter
    void display(Test obj)
    {
        System.out.println("a = " + a + "  b = " + b);
    }

    //Method that returns current class instance
    void get()
    {
        display(this);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get();
    }
}
```

Output:

```
a = 10  b = 20
```

This article is contributed by Mehak Narang.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

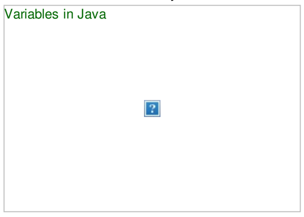
See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# Variables in Java

A variable is the name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before they can be used.

**How to declare variables?**
We can declare variables in java as follows:



**datatype**: Type of data that can be stored in this variable.
**variable_name**: Name given to the variable.
**value**: It is the initial value stored in the variable.

**Examples**:

```
float simpleInterest; //Declaring float variable
int time = 10, speed = 20; //Declaring and Initializing integer variable
char var = 'h'; // Declaring and Initializing character variable
```

**Types of variables**

There are three types of variables in Java:

- Local Variables
- Instance Variables
- Static Variables

Let us now learn about each one of these variables in detail.

1. **Local Variables**: A variable defined within a block or method or constructor is called local variable.
   - These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
   - The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.

   **Sample Program 1:**

   ```
   public class StudentDetails
   {
       public void StudentAge()
       {   //local variable age
        int age = 0;
        age = age + 5;
        System.out.println("Student age is : " + age);
       }

       public static void main(String args[])
       {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
       }
   }
   ```

   Output:

   ```
   Student age is : 5
   ```

   In the above program the variable age is local variable to the function StudentAge(). If we use the variable age outside StudentAge() function, the compiler will produce an error as shown in below program.

   **Sample Program 2:**

   ```
   public class StudentDetails
   {
       public void StudentAge()
       {   //local variable age
        int age = 0;
        age = age + 5;
       }

       public static void main(String args[])
       {
           //using local variable age outside it's scope
        System.out.println("Student age is : " + age);
       }
   }
   ```

   Output:

   ```
   error: cannot find symbol
    " + age);
   ```

2. **Instance Variables**: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
    - As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
    - Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.

    **Sample Program**:

```
import java.io.*;
class Marks
{
    //These variables are instance variables.
    //These variables are in a class and are not inside any function
    int engMarks;
    int mathsMarks;
    int phyMarks;
}

class MarksDemo
{
    public static void main(String args[])
    {   //first object
        Marks obj1 = new Marks();
        obj1.engMarks = 50;
        obj1.mathsMarks = 80;
        obj1.phyMarks = 90;

        //second object
        Marks obj2 = new Marks();
        obj2.engMarks = 80;
        obj2.mathsMarks = 60;
        obj2.phyMarks = 85;

        //displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.engMarks);
        System.out.println(obj1.mathsMarks);
        System.out.println(obj1.phyMarks);

        //displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.engMarks);
        System.out.println(obj2.mathsMarks);
        System.out.println(obj2.phyMarks);
    }
}
```

    Output:

```
Marks for first object:
50
80
90
Marks for second object:
80
60
85
```

    As you can see in the above program the variables, *engMarks* , *mathsMarks* , *phyMarks*are instance variables. In case we have multiple objects as in the above program, each object will have its own copies of instance variables. It is clear from the above output that each object will have its own copy of instance variable.

3. **Static Variables**: Static variables are also known as Class variables.
    - These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
    - Unlike instance variables, we can only have only copy of a static variable per class irrespective of how many objects we create.
    - Static variables are created at start of program execution and destroyed automatically when execution ends.

    To access static variables, we need not to create any object of that class, we can simply access the variable as:

```
class_name.variable_name;
```

    **Sample Program**:

```
import java.io.*;
class Emp {

   // static variable salary
   public static double salary;
   public static String name = "Harsh";
}

public class EmpDemo
{
     public static void main(String args[]) {

      //accessing static variable without object
      Emp.salary = 1000;
      System.out.println(Emp.name + "'s average salary:" + Emp.salary);
   }

}
```

    output:

```
Harsh's average salary:1000.0
```

**Instance variable Vs Static variable**

- Each object will have its **own copy** of instance variable whereas We can only have **one copy** of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will **not be reflected** in other objects as each object has its own copy of instance variable. In case of static, changes **will be reflected** in other objects as static variables are common to all object of a class.
- We can access instance variables **through object references** and Static Variables can be accessed **directly using class name.**
- Syntax for static and instance variables:

```
class Example
    {
        static int a; //static variable
        int b;       //instance variable
    }
```

**Similar Articles:**

- Scope of Variables in Java
- Comparison of static keyword in C++ and Java
- Are static local variables allowed in Java?
- Instance Variable Hiding in Java

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Basics

# How are Java objects stored in memory?

In Java, all objects are dynamically allocated on Heap. This is different from C++ where objects can be allocated memory either on Stack or on Heap. In C++, when we allocate abject using new(), the abject is allocated on Heap, otherwise on Stack if not global or static.

In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap (See this for more details).

For example, following program fails in compilation. Compiler gives error *"Error here because t is not initialed"*.

```
class Test {
    // class contents
    void show() {
        System.out.println("Test::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Test t;
        t.show(); // Error here because t is not initialed
    }
}
```

Allocating memory using new() makes above program work.

```
class Test {
    // class contents
    void show() {
        System.out.println("Test::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Test t = new Test(); //all objects are dynamically allocated
        t.show(); // No error
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# Scope of Variables In Java

Scope of a variable is the part of the program where the variable is accessible. Like C/C++, in Java, all identifiers are lexically (or statically) scoped, i.e.scope of a variable can determined at compiler time and independent of function call stack.
Java programs are organized in the form of classes. Every class is part of some package. Java scope rules can be covered under following categories.

**Member Variables (Class Level Scope)**

These variables must be declared inside class (outside any function). They can be directly accessed anywhere in class. Let's take a look at an example:

```
public class Test
{
    // All variables defined directly inside a class
    // are member variables
    int a;
    private String b
    void method1() {....}
    int method2() {....}
    char c;
}
```

- We can declare class variables anywhere in class, but outside methods.
- Access specified of member variables doesn't effect scope of them within a class.
- Member variables can be accessed outside a class with following rules

```
Modifier     Package  Subclass  World

public        Yes      Yes     Yes

protected      Yes      Yes     No

Default (no
modifier)     Yes      No     No

private       No       No     No
```

**Local Variables (Method Level Scope)**

Variables declared inside a method have method level scope and can't be accessed outside the method.

```
public class Test
{
    void method1()
    {
        // Local variable (Method level scope)
        int x;
    }
}
```

**Note :** Local variables don't exist after method's execution is over.

Here's another example of method scope, except this time the variable got passed in as a parameter to the method:

```
class Test
{
    private int x;
    public void setX(int x)
    {
        this.x = x;
    }
}
```

The above code uses this keyword to differentiate between the local and class variables.

As an exercise, predict the output of following Java program.

```
public class Test
{
    static int x = 11;
    private int y = 33;
    public void method1(int x)
    {
        Test t = new Test();
        this.x = 22;
        y = 44;

        System.out.println("Test.x: " + Test.x);
        System.out.println("t.x: " + t.x);
        System.out.println("t.y: " + t.y);
        System.out.println("y: " + y);
    }

    public static void main(String args[])
    {
        Test t = new Test();
        t.method1(5);
    }
}
```

**Output:**

```
Test.x: 22
t.x: 22
t.y: 33
y: 44
```

**Loop Variables (Block Scope)**

A variable declared inside pair of brackets "{" and "}" in a method has scope withing the brackets only.

```
public class Test
{
    public static void main(String args[])
    {
        {
            // The variable x has scope within
            // brackets
            int x = 10;
            System.out.println(x);
        }

        // Uncommenting below line would produce
        // error since variable x is out of scope.

        // System.out.println(x);
    }
}
```

Output:

```
10
```

As another example, consider following program with a for loop.

```
class Test
{
    public static void main(String args[])
    {
        for (int x = 0; x < 4; x++)
        {
            System.out.println(x);
        }

        // Will produce error
        System.out.println(x);
    }
}
```

Output:

```
11: error: cannot find symbol
        System.out.println(x);                    ^
```

The right way of doing above is,

```
// Above program after correcting the error
class Test
{
    public static void main(String args[])
    {
        int x;
        for (x = 0; x < 4; x++)
        {
            System.out.println(x);
        }
        System.out.println(x);
    }
}
```

**Output:**

```
0
1
2
3
4
```

Let's look at tricky example of loop scope. Predict the output of following program. You may be surprised if you are regular C/C++ programmer.

```
class Test
{
    public static void main(String args[])
    {
        int a = 5;
        for (int a = 0; a < 5; a++)
        {
            System.out.println(a);
        }
    }
}
```

Output :

```
6: error: variable a is already defined in method go(int)
       for (int a = 0; a
```

A similar program in C++ works.  See this.

As an exercise, predict the output of following Java program.

```
class Test
{
    public static void main(String args[])
    {
        {
            int x = 5;
            {
                int x = 10;
                System.out.println(x);
            }
        }
    }
}
```

**Some Important Points about Variable scope in Java:**

- In general, a set of curly brackets { } defines a scope.
- In Java we can usually access a variable as long as it was defined within the same set of brackets as the code we are writing or within any curly brackets inside of the curly brackets where the variable was defined.
- Any variable defined in a class outside of any method can be used by all member methods.
- When a method has same local variable as a member, this keyword can be used to reference the current class variable.
- For a variable to be read after the termination of a loop, It must be declared before the body of the loop.

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Java

## Do we need forward declarations in Java?

Predict output of the following Java program.

```
// filename: Test2.java

// main() function of this class uses Test1 which is declared later in
// this file
class Test2 {
    public static void main(String[] args) {
        Test1 t1 = new Test1();
        t1.fun(5);
    }
}
class Test1 {
    void fun(int x) {
        System.out.println("fun() called: x = " + x);
    }
}
```

Output:

```
fun() called: x = 5
```

The Java program compiles and runs fine. Note that *Test1* and *fun()* are not declared before their use. Unlike C++, we don't need forward declarations in Java. Identifiers (class and method names) are recognized automatically from source files. Similarly, library methods are directly read from the libraries, and there is no need to create header files with declarations. Java uses naming scheme where package and public class names must follow directory and file names respectively. This naming scheme allows Java compiler to locate library files.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Widening Primitive Conversion in Java

Here is a small code snippet given. Try to Guess the output

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.print("Y" + "O");
        System.out.print('L' + 'O');
    }
}
```

*At first glance, we expect “YOLO” to be printed.*

**Actual Output:**
“YO155”.

**Explanation:**
When we use double quotes, the text is treated as a string and “YO” is printed, but when we use single quotes, the characters ‘L’ and ‘O’ are converted to int. This is called widening primitive conversion. After conversion to integer, the numbers are added ( ‘L’ is 76 and ‘O’ is 79) and 155 is printed.

Now try to guess the output of following:

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.print("Y" + "O");
        System.out.print('L');
        System.out.print('O');
    }
}
```

**Output:** YOLO

**Explanation:** This will now print “YOLO” instead of “YO7679”. It is because the widening primitive conversion happens only when ‘+’ operator is present.

Widening primitive conversion is applied to convert either or both operands as specified by the following rules. The result of adding Java chars, shorts or bytes is an **int**:

- If either operand is of type double, the other is converted to double.
- Otherwise, if either operand is of type float, the other is converted to float.
- Otherwise, if either operand is of type long, the other is converted to long.
- Otherwise, **both operands are converted to type int**

Reference: http://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.6.2

This article is contributed by **Anurag Rai**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**GATE CS Corner    Company Wise Coding Practice**

## Numbers in Java (With 0 Prefix and with Strings)

Consider the following Java program.

```
import java.io.*;
class GFG
{
    public static void main (String[] args)
    {
        int x = 012;
        System.out.print(x);
    }
}
```

Output:

```
10
```

The reason for above output is, when a 0 is prefixed the value is considered octal, since 12 in octal is 10 in decimal, the result is 10. Similarly, if i = 0112, result will be 74 (in decimal). This behavior is same as C/C++ (see this).

Also,

```
import java.io.*;
class GFG
{
    public static void main (String[] args)
    {
        String s = 3 + 2 + "hello" + 6 + 4;
        System.out.print(s);
    }
}
```

Output :

```
5hello64
```

Java takes the numbers before the strings are introduced as int and once the string literals are introduced, all the following numbers are considered as strings.

This article is contributed by **Hiresh Trivedi**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

## final variables in Java

In Java, when final keyword is used with a variable of primitive data types (int, float, .. etc), value of the variable cannot be changed.

For example following program gives error because i is final.

```
public class Test {
    public static void main(String args[]) {
        final int i = 10;
        i = 30; // Error because i is final.
    }
}
```

When final is used with non-primitive variables (Note that non-primitive variables are always references to objects in Java), the members of the referred object can be changed. final for non-primitive variables just mean that they cannot be changed to refer to any other object

```
class Test1 {
   int i = 10;
}

public class Test2 {
    public static void main(String args[]) {
      final Test1 t1 = new Test1();
      t1.i = 30;  // Works
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## volatile keyword in Java

Using volatile is yet another way (like synchronized, atomic wrapper) of making class thread safe. Thread safe means that a method or class instance can be used by multiple threads at the same time without any problem.

Consider below simple example.

```
class SharedObj
{
   // Changes made to sharedVar in one thread
   // may not immediately reflect in other thread
   static int sharedVar = 6;
}
```

Suppose that two threads are working on **SharedObj**. If two threads run on different processors each thread may have its own local copy of **sharedVar**. If one thread modifies its value the change might not reflect in the original one in the main memory instantly. This depends on the write policy of cache. Now the other thread is not aware of the modified value which leads to data inconsistency.

Below diagram shows that if two threads are run on different processors, then value of **sharedVar** may be different in different threads.

volatileJava

Note that write of normal variables without any synchronization actions, might not be visible to any reading thread (this behavior is called sequential consistency). Although most modern hardware provide good cache coherence therefore most probably the changes in one cache are reflected in other but it's not a good practice to rely on hardware for to 'fix' a faulty application.

```
class SharedObj
{
   // volatile keyword here makes sure that
   // the changes made in one thread are
   // immediately reflect in other thread
   static volatile int sharedVar = 6;
}
```

Note that volatile should not be confused with static modifier. static variables are class members that are shared among all objects. There is only one copy of them in main memory.

**volatile vs synchronized:**
Before we move on let's take a look at two important features of locks and synchronization.

1. **Mutual Exclusion:** It means that only one thread or process can execute a block of code (critical section) at a time.
2. **Visibility**: It means that changes made by one thread to shared data are visible to other threads.

Java's synchronized keyword guarantees both mutual exclusion and visibility. If we make the blocks of threads that modifies the value of shared variable synchronized only one thread can enter the block and changes made by it will be reflected in the main memory. All other thread trying to enter the block at the same time will be blocked and put to sleep.

In some cases we may only desire the visibility and not atomicity. Use of synchronized in such situation is an overkill and may cause scalability problems. Here volatile comes to the rescue. Volatile variables have the visibility features of synchronized but not the atomicity features. The values of volatile variable will never be cached and all writes and reads will be done to and from the main memory. However, use of volatile is limited to very restricted set of cases as most of the times atomicity is desired. For example a simple increment statement such as x = x + 1; or x++ seems to be a single operation but is s really a compound read-modify-write sequence of operations that must execute atomically.

**volatile in Java vs C/C++:**

Volatile in java is different from "volatile" qualifier in C/C++. For Java, "volatile" tells the compiler that the value of a variable must never be cached as its value may change outside of the scope of the program itself. In C/C++, "volatile" is needed when developing embedded systems or device drivers, where you need to read or write a memory-mapped hardware device. The contents of a particular device register could change at any time, so you need the "volatile" keyword to ensure that such accesses aren't optimized away by the compiler.

**References:**

https://www.ibm.com/developerworks/java/library/j-jtp06197/
https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html
http://tutorials.jenkov.com/java-concurrency/volatile.html
https://pveentjer.wordpress.com/2008/05/17/jmm-thank-god-or-the-devil-for-strong-cache-coherence/

This artcile is contributed by Sulabh Kumar. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Instance Variable Hiding in Java

In Java, if there a local variable in a method with same name as instance variable, then the local variable hides the instance variable. If we want to reflect the change made over to the instance variable, this can be achieved with the help of this reference.

```
class Test
{
    // Instance variable or member variable
    private int value = 10;

    void method()
    {
        // This local variable hides instance variable
        int value = 40;

        System.out.println("Value of Instance variable :"
                            + this.value);
        System.out.println("Value of Local variable :"
                            + value);
    }
}

class UseTest
{
    public static void main(String args[])
    {
        Test obj1 = new Test();
        obj1.method();
    }
}
```

Output:

```
Value of Instance variable :10
Value of Local variable :40
```

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Using underscore in Numeric Literals in Java

A new feature was introduced by JDK 7 which allows to write numeric literals using the underscore character. Numeric literals are broken to enhance the readability.

This feature enables us to separate groups of digits in numeric literals, which improves readability of code. For instance, if our code contains numbers with many digits, we can use an underscore character to separate digits in groups of three, similar to how we would use a punctuation mark like a comma, or a space, as a separator.

The following example shows different ways we can use underscore in numeric literals:

```
// Java program to demonstrate that we can use underscore
// in numeric literals
class Test
{
     public static void main (String[] args)
               throws java.lang.Exception
     {
         int inum = 1_00_00_000;
         System.out.println("inum:" + inum);

         long lnum = 1_00_00_000;
         System.out.println("lnum:" + lnum);

         float fnum = 2.10_001F;
         System.out.println("fnum:" + fnum);

         double dnum = 2.10_12_001;
         System.out.println("dnum:" + dnum);
      }
}
```

**Output:**

```
inum: 10000000
lnum: 10000000
fnum: 2.10001
dnum: 2.1012001
```

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

# Type conversion in Java with Examples

When you assign value of one data type to another, the two types might not be compatible with each other. If the data types are compatible, then Java will perform the conversion automatically known as Automatic Type Conversion and if not then they need to be casted or converted explicitly. For example, assigning an int value to a long variable.

**Widening or Automatic Type Conversion**

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

Widening or Automatic Type Conversion

Example:

```
class Test
{
 public static void main(String[] args)
 {
  int i = 100;

  //automatic type conversion
  long l = i;

  //automatic type conversion
  float f = l;
  System.out.println("Int value "+i);
  System.out.println("Long value "+l);
  System.out.println("Float value "+f);
 }
}
```

Output:

```
Int value 100
Long value 100
Float value 100.0
```

**Narrowing or Explicit Conversion**

If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

Narrowing or Explicit Conversion

char and number are not compatible with each other. Let’s see when we try to convert one into other.

```
//Java program to illustrate incompatible data
// type for explicit type conversion
public class Test
{
  public static void main(String[] argv)
  {
    char ch = 'c';
    int num = 88;
    ch = num;
  }
}
```

Error:

```
7: error: incompatible types: possible lossy conversion from int to char
    ch = num;
         ^
1 error
```

**How to do Explicit Conversion?**

Example:

```
//Java program to illustrate explicit type conversion
class Test
{
 public static void main(String[] args)
 {
  double d = 100.04;

  //explicit type casting
  long l = (long)d;

  //explicit type casting
  int i = (int)l;
  System.out.println("Double value "+d);

  //fractional part lost
  System.out.println("Long value "+l);

  //fractional part lost
  System.out.println("Int value "+i);
 }
}
```

Output:

```
Double value 100.04
Long value 100
Int value 100
```

While assigning value to byte type the fractional part is lost and is reduced to modulo 256(range of byte).
Example:

```
//Java program to illustrate Conversion of int and double to byte
class Test
{
 public static void main(String args[])
 {
  byte b;
  int i = 257;
  double d = 323.142;
  System.out.println("Conversion of int to byte.");

  //i%256
  b = (byte) i;
  System.out.println("i = b " + i + " b = " + b);
  System.out.println("\nConversion of double to byte.");

  //d%256
  b = (byte) d;
  System.out.println("d = " + d + " b= " + b);
 }
}
```

Output:

```
Conversion of int to byte.
i = 257 b = 1

Conversion of double to byte.
d = 323.142 b = 67
```

**Type promotion in Expressions**

While evaluating expressions, the intermediate value may exceed the range of operands and hence the expression value will be promoted. Some conditions for type promotion are:

1. Java automatically promotes each byte, short, or char operand to int when evaluating an expression.
2. If one operand is a long, float or double the whole expression is promoted to long, float or double respectively.

Example:

```
//Java program to illustrate Type promotion in Expressions
class Test
{
 public static void main(String args[])
 {
  byte b = 42;
  char c = 'a';
  short s = 1024;
  int i = 50000;
  float f = 5.67f;
  double d = .1234;

  // The Expression
  double result = (f * b) + (i / c) - (d * s);

  //Result after all the promotions are done
  System.out.println("result = " + result);
 }
}
```

Output:

```
Result = 626.7784146484375
```

**Explicit type casting in Expressions**

While evaluating expressions, the result is automatically updated to larger data type  of the operand. But if we store that result in any smaller data type it generates compile time error, due to which we need to type cast the result.
Example:

```
//Java program to illustrate type casting int to byte
class Test
{
 public static void main(String args[])
 {
  byte b = 50;

  //type casting int to byte
  b = (byte)(b * 2);
  System.out.println(b);
 }
}
```

Output

```
 100
```

NOTE- In case of single operands the result gets converted to int and then it is type casted accordingly.
Example:

```
//Java program to illustrate type casting int to byte
class Test
{
    public static void main(String args[])
    {
        byte b = 50;
        //type casting int to byte
        b = (byte)(b * 2);
        System.out.println(b);
    }
}
```

Output

```
100
```

This article is contributed by **Apoorva singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Bitwise right shift operators in Java

In C/C++ there is only one right shift operator '>>' which should be used only for positive integers or unsigned integers. Use of right shift operator for negative numbers is not recommended in C/C++, and when used for negative numbers, output is compiler dependent (See this). Unlike C++, Java supports following two right shift operators.

**1) >> (Signed right shift)** In Java, the operator '>>' is signed right shift operator. All integers are signed in Java, and it is fine to use >> for negative numbers. The operator '>>' uses the sign bit (left most bit) to fill the trailing positions after shift. If the number is negative, then 1 is used as a filler and if the number is positive, then 0 is used as a filler. For example, if binary representation of number is **1**0….100, then right shifting it by 2 using >> will make it **11**…….1.
See following Java programs as example '>>'

```
class Test {
    public static void main(String args[])  {
       int x = -4;
       System.out.println(x>>1);
       int y = 4;
       System.out.println(y>>1);
    }
}
```

Output:

```
-2
2
```

**2) >>> (Unsigned right shift)** In Java, the operator '>>>' is unsigned right shift operator. It always fills 0 irrespective of the sign of the number.

```
class Test {
    public static void main(String args[])  {

       // x is stored using 32 bit 2's complement form.
       // Binary representation of -1 is all 1s (111..1)
       int x = -1;

       System.out.println(x>>>29);  // The value of 'x>>>29' is 00...0111
       System.out.println(x>>>30);  // The value of 'x>>>30' is 00...0011
       System.out.println(x>>>31);  // The value of 'x>>>31' is 00...0001
   }
}
```

Output:

```
7
3
1
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Comparison of Autoboxed Integer objects in Java

When we assign an integer value to an Integer object, the value is autoboxed into an Integer object. For example the statement "Integer x = 10" creates an object 'x' with value 10.

Following are some interesting output questions based on comparison of Autoboxed Integer objects.

Predict the output of following Java Program

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
         Integer x = 400, y = 400;
         if (x == y)
            System.out.println("Same");
         else
            System.out.println("Not Same");
    }
}
```

Output:

```
Not Same
```

Since x and y refer to different objects, we get the output as "Not Same"

The output of following program is a surprise from Java.

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
         Integer x = 40, y = 40;
         if (x == y)
            System.out.println("Same");
```

```
        else
           System.out.println("Not Same");
    }
}
```

Output:

```
Same
```

In Java, values from -128 to 127 are cached, so the same objects are returned. The implementation of valueOf() uses cached objects if the value is between -128 to 127.

If we explicitly create Integer objects using new operator, we get the output as “Not Same”. See the following Java program. In the following program, valueOf() is not used.

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
         Integer x = new Integer(40), y = new Integer(40);
        if (x == y)
           System.out.println("Same");
        else
           System.out.println("Not Same");
    }
}
```

Output:

```
Not Same
```

This article is compiled by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# Addition and Concatenation in Java

**Try to predict the output of following code:**

```
public class Geeksforgeeks
{
    public static void main(String[] args)
    {
        System.out.println(2+0+1+6+"GeeksforGeeks");
        System.out.println("GeeksforGeeks"+2+0+1+6);
        System.out.println(2+0+1+5+"GeeksforGeeks"+2+0+1+6);
        System.out.println(2+0+1+5+"GeeksforGeeks"+(2+0+1+6));
    }
}
```

**Explanation:**
The output is

```
9GeeksforGeeks
GeeksforGeeks2016
8GeeksforGeeks2016
8GeeksforGeeks9
```

This unpredictable output is due the fact that the compiler evaluates the given expression from left to right given that the operators have same precedence. Once it encounters the String, it considers the rest of the expression as of a String (again based on the precedence order of the expression).

- **System.out.println(2 + 0 + 1 + 6 + “GeeksforGeeks”);** *// It prints the addition of 2,0,1 and 6 which is equal to 9*
- **System.out.println(“GeeksforGeeks” + 2 + 0 + 1 + 6);** *//It prints the concatenation of 2,0,1 and 6 which is 2016  since it encounters the string initially. Basically, Strings take precedence because they have a higher casting priority than integers do.*
- **System.out.println(2 + 0 + 1 + 5 + “GeeksforGeeks” + 2 + 0 + 1 + 6);** *//It prints the addition of 2,0,1 and 5 while the concatenation of 2,0,1 and 6 based on the above given examples.*
- **System.out.println(2 + 0 + 1 + 5 + “GeeksforGeeks” + (2 + 0 + 1 + 6));** *//It prints the addition of  both 2,0,1 and 5  and 2,0,1 and 6 based due the precedence of ( ) over +. Hence expression in ( ) is calculated first and then the further evaluation takes place.*

This article is contributed by **Pranjal Mathur.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# Java Numeric Promotion in Conditional Expression

The conditional operator ? : uses the boolean value of one expression to decide which of two other expressions should be evaluated.

So, we expect the expression,

```
Object o1 = true ? new Integer(4) : new Float(2.0));
```

**to be same as,**

```
Object o2;
if (true)
    o2 = new Integer(4);
else
    o2 = new Float(2.0);
```

But the result of running the code gives an unexpected result.

```
// A Java program to demonstrate that we should be careful
// when replacing conditional operator with if else or vice
// versa
import java.io.*;
class GFG
```

```
{
    public static void main (String[] args)
    {
        // Expression 1 (using ?: )
        // Automatic promotion in conditional expression
        Object o1 = true ? new Integer(4) : new Float(2.0);
        System.out.println(o1);

        // Expression 2 (Using if-else)
        // No promotion in if else statement
        Object o2;
        if (true)
            o2 = new Integer(4);
        else
            o2 = new Float(2.0);
        System.out.println(o2);
    }
}
```

Output:

```
4.0
4
```

According to Java Language Specification Section 15.25, the conditional operator will implement numeric type promotion if there are two different types as 2nd and 3rd operand.  The rules of conversion are defined at Binary Numeric Promotion. Therefore, according to the rules given, If either operand is of type double, the other is converted to double and hence 4 becomes 4.0.

Whereas, the if/else construct does not perform numeric promotion and hence behaves as expected.

This article is contributed by **Deepak Garg**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

---

# String Class in Java

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

**Creating a String**

There are two ways to create string in Java:

- ***String literal***

  ```
  String s = “GeeksforGeeks”;
  ```

- **Using *new* keyword**

  ```
  String s = new String (“GeeksforGeeks”);
  ```

**String Methods**

- **int length():** Returns the number of characters in the String.

  ```
  "GeeksforGeeks".length();  // returns 13
  ```

- **Char charAt(int i):** Returns the character at i^th index.

  ```
  "GeeksforGeeks".charAt(3); // returns  ‘k’
  ```

- **String substring (int i):** Return the substring from the i^th  index character to end.

  ```
  "GeeksforGeeks".substring(3); // returns “ksforGeeks”
  ```

- **String substring (int i, int j):** Returns the substring from i to j-1 index.

  ```
  "GeeksforGeeks".substring(2, 5); // returns “eks”
  ```

-  **String concat( String str):** Concatenates specified string to the end of this string.

  ```
  String s1 = ”Geeks”;
  String s2 = ”forGeeks”;
  String output = s1.concat(s2); // returns “GeeksforGeeks”
  ```

- **int indexOf (String s):** Returns the index within the string of the first occurrence of the specified string.

  ```
  String s = ”Learn Share Learn”;
  int output = s.indexOf(“Share”); // returns 6
  ```

- **int indexOf (String s, int i):** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.

  ```
  String s = ”Learn Share Learn”;
  int output = s.indexOf(‘a’,3);// returns 8
  ```

- **Int lastindexOf( int ch):** Returns the index within the string of the last occurrence of the specified string.

  ```
  String s = ”Learn Share Learn”;
  int output = s.lastindexOf(‘a’); // returns 14
  ```

- **boolean equals( Object otherObj):** Compares this string to the specified object.

  ```
  Boolean out = “Geeks”.equals(“Geeks”); // returns true
  Boolean out = “Geeks”.equals(“geeks”); // returns false
  ```

- **boolean  equalsIgnoreCase (String anotherString):** Compares string to another string, ignoring case considerations.

  ```
  Boolean out= “Geeks”.equalsIgnoreCase(“Geeks”); // returns true
  Boolean out = “Geeks”.equalsIgnoreCase(“geeks”); // returns true
  ```

-  **int compareTo( String anotherString):** Compares two string lexicographically.

  ```
  int out = s1.compareTo(s2);  // where s1 ans s2 are
                          // strings to be compared

  This returns difference s1-s2. If :
  out < 0  // s1 comes before s2
  ```

```
out = 0  // s1 and s2 are equal.
out >0   // s1 comes after s2.
```

- **int compareToIgnoreCase( String anotherString):** Compares two string lexicographically, ignoring case considerations.

```
int out = s1.compareTo(s2);  // where s1 ans s2 are
                             // strings to be compared

This returns difference s1-s2. If :
out < 0  // s1 comes before s2
out = 0   // s1 and s2 are equal.
out >0   // s1 comes after s2.
```

*Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).*

- **String toLowerCase():** Converts all the characters in the String to lower case.

```
String word1 = “HeLLo”;
String word3 = word1.toLowerCase(); // returns “hello"
```

- **String toUpperCase():** Converts all the characters in the String to upper case.

```
String word1 = “HeLLo”;
String word2 = word1.toUpperCase(); // returns “HELLO”
```

- **String trim():** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.

```
String word1 = “ Learn Share Learn “;
String word2 = word1.trim(); // returns “Learn Share Learn”
```

- **String replace (char oldChar, char newChar):** Returns new string by replacing all occurrences of *oldChar* with *newChar.*

```
String s1 = “feeksforfeeks“;
String s2 = “feeksforfeeks”.replace(‘f’ ,’g’); // returns “geeksgorgeeks”
```

*Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks*

Program to illustrate all string  methods:

```
// Java code to illustrate different constructors and methods
// String class.

import java.io.*;
import java.util.*;
class Test
{
    public static void main (String[] args)
    {
        String s= "GeeksforGeeks";
        // or String s= new String ("GeeksforGeeks");

        // Returns the number of characters in the String.
        System.out.println("String length = " + s.length());

        // Returns the character at ith index.
        System.out.println("Character at 3rd position = "
                       + s.charAt(3));

        // Return the substring from the ith  index character
        // to end of string
        System.out.println("Substring " + s.substring(3));

        // Returns the substring from i to j-1 index.
        System.out.println("Substring  = " + s.substring(2,5));

        // Concatenates string2 to the end of string1.
        String s1 = "Geeks";
        String s2 = "forGeeks";
        System.out.println("Concatenated string  = " +
                        s1.concat(s2));

        // Returns the index within the string
        // of the first occurrence of the specified string.
        String s4 = "Learn Share Learn";
        System.out.println("Index of Share " +
                       s4.indexOf("Share"));

        // Returns the index within the string of the
        // first occurrence of the specified string,
        // starting at the specified index.
        System.out.println("Index of a  = " +
                       s4.indexOf('a',3));

        // Checking equality of Strings
        Boolean out = "Geeks".equals("geeks");
        System.out.println("Checking Equality  " + out);
        out = "Geeks".equals("Geeks");
        System.out.println("Checking Equality  " + out);

        out = "Geeks".equalsIgnoreCase("gEeks ");
        System.out.println("Checking Equality" + out);

        int out1 = s1.compareTo(s2);
        System.out.println("If s1 = s2" + out);

        // Converting cases
        String word1 = "GeeKyMe";
        System.out.println("Changing to lower Case " +
                        word1.toLowerCase());

        // Converting cases
        String word2 = "GeekyME";
        System.out.println("Changing to UPPER Case " +
                        word1.toUpperCase());

        // Trimming the word
        String word4 = " Learn Share Learn ";
        System.out.println("Trim the word " + word4.trim());

        // Replacing characters
        String str1 = "feeksforfeeks";
        System.out.println("Original String " + str1);
        String str2 = "feeksforfeeks".replace('f' ,'g') ;
        System.out.println("Replaced f with g -> " + str2);
    }
}
```

Output :

```
String length = 13
Character at 3rd position = k
Substring ksforGeeks
Substring = eks
Concatenated string = GeeksforGeeks
Index of Share 6
Index of a = 8
Checking Equality false
Checking Equality true
Checking Equalityfalse
If s1 = s2false
Changing to lower Case geekyme
Changing to UPPER Case GEEKYME
Trim the word Learn Share Learn
Original String feeksforfeeks
Replaced f with g -> geeksgorgeeks
```

This article is contributed by **Rahul Agrawal.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# How to Initialize and Compare Strings in Java?

**Initializing Strings in Java**

**1. Direct Initialization(String Constant)** : In this method, a String constant object will be created in String pooled area which is inside heap area in memory. As it is a **constant**, we can't modify it,i.e. String class is **immutable**.

**Examples:**

```
String str = "GeeksForGeeks";

str = "geeks"; // This statement will make str
             // point to new String constant("geeks")
             // rather then modifying the previous
             // String constant.
```

**String str = "GeeksForGeeks";**

as

**str = "geeks";**

as

**Note:** If we again write **str = "GeeksForGeeks"** as next line, then it first check that if given String constant is present in String pooled area or not. If it present then str will point to it, otherwise creates a new String constant.

**2. Object Initialization (Dynamic)**: In this method, a String object will be created in heap area (not inside String pooled area as in upper case). We can modify it.Also with same value,a String constant is also created in String pooled area,but the variable will point to String object in heap area only.

**Examples:**

```
String str = new String("very");
str = "good";
```

**String str = new String("very")**

as

**str = "good"**

Now this is a direct assignment,so String constant with value "good" is created in String pooled area and str will point to that.

as

**Note:** If we again write **str = new String("very")**, then it will create a new object with value "very", rather than pointing to the available objects in heap area with same value.But if we write **str = "very"**,then it will point to String constant object with value "very",present in String pooled area.

**Comparing Strings and their References**

**1. equals() method:** It compares **values** of string for equality. Return type is boolean. In almost all the situation you can use useObjects.equals().

**2. == operator:** It compares **references not values**. Return type is boolean. == is used in rare situations where you know you're dealing with interned strings.

**3. compareTo() method:** It compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.For example,if str1 and str2 are two string variables then if

**str1 == str2 :** return 0
**str1 > str2 :** return a positive value
**str1 < str2 :** return a negative value

**Note:** The positive and negative values returned by **compareTo** method is the difference of first unmatched character in the two strings.

```
// Java program to show how to compare Strings

public class Test
{
    public static void main(String[] args)
    {
        String s1 = "Ram";
        String s2 = "Ram";
        String s3 = new String(" Ram");
        String s4 = new String(" Ram");
        String s5 = "Shyam";
        String nulls1 = null;
        String nulls2 = null;

        System.out.println(" Comparing strings with equals:");
        System.out.println(s1.equals(s2));
        System.out.println(s1.equals(s3));
        System.out.println(s1.equals(s5));

        System.out.println(" Comparing strings with ==:");
        System.out.println(s1==s2);
        System.out.println(s1==s3);
        System.out.println(s3==s4);

        System.out.println(" Comparing strings with compareto:");
        System.out.println(s1.compareTo(s3));
        System.out.println(s1.compareTo(s5));
    }
}
```

Output:

```
Comparing strings with equals:
true
true
false
Comparing strings with ==:
true
false
false
Comparing strings with compareto:
0
-1
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

## Different ways for Integer to String Conversions In Java

1. **Convert using Integer.toString(int)**
   The Integer class has a static method that returns a String object representing the specified int parameter.
   **Syntax :**

   ```
   public static String toString(int i)
   ```

   The argument *i* is converted and returned as a string instance. If the number is negative, the sign will be preserved.
   **Example :**

   ```
   class GfG
   {
     public static void main(String args[])
     {
       int a = 1234;
       int b = -1234;
       String str1 = Integer.toString(a);
       String str2 = Integer.toString(b);
       System.out.println("String str1 = " + str1);
       System.out.println("String str2 = " + str2);
     }
   }
   ```

   Output:

   ```
   String str1 = 1234
   String str2 = -1234
   ```

2. **Convert using String.valueOf(int)**
   **Example :**

   ```
   class GfG
   {
     public static void main(String args[])
     {
       int c = 1234;
       String str3 = String.valueOf(c);
       System.out.println("String str3 = " + str3);
     }
   }
   ```

   or

   ```
   class GfG
   {
     public static void main(String args[])
     {
       String str3 = String.valueOf(1234);
       System.out.println("String str3 = " + str3);
     }
   }
   ```

   Output:

   ```
   String str3 = 1234
   ```

3. **Convert using Integer(int).toString()**
   This methods uses instance of Integer class to invoke it's toString() method.
   **Example :**

   ```
   class GfG
   {
     public static void main(String args[])
     {
       int d = 1234;
       Integer obj = new Integer(d);
       String str4 = obj.toString();
       System.out.println("String str4 = " + str4);
     }
   }
   ```

   or

   ```
   class GfG
   {
     public static void main(String args[])
     {
       int d = 1234;
       String str4 = new Integer(d).toString();
       System.out.println("String str4 = " + str4);
     }
   }
   ```

or

```
class GfG
{
  public static void main(String args[])
  {
    String str4 = new Integer(1234).toString();
    System.out.println("String str4 = " + str4);
  }
}
```

Output:

```
String str4 = 1234
```

If your variable is of primitive type (int), it is better to use Integer.toString(int) or String.valueOf(int). But if your variable is already an instance of Integer (wrapper class of the primitive type int), it is better to just invoke it's toString() method as shown above.
**This method is not efficient as instance of Integer class is created before conversion is performed.**

4. **Convert using DecimalFormat**
   The class **java.text.DecimalFormat** is a class that formats a number to a String.
   **Example :**

```
import java.text.DecimalFormat;
class GfG
{
  public static void main(String args[])
  {
    int e = 12345;
    DecimalFormat df = new DecimalFormat("#");
    String str5 = df.format(e);
    System.out.println(str5);
  }
}
```

Output:

```
String str5 = 12345
```

or

```
import java.text.DecimalFormat;
class GfG
{
  public static void main(String args[])
  {
    int e = 12345;
    DecimalFormat df = new DecimalFormat("#,###");
    String Str5 = df.format(e);
    System.out.println(Str5);
  }
}
```

Output:

```
String str5 = 12,345
```

Using this method, you can specify the number of decimal places and comma separator for readability.

5. **Convert using StringBuffer or StringBuilder**
   StringBuffer is a class that is used to concatenate multiple values into a String. StringBuilder works similarly but not thread safe like StringBuffer.
   **StringBuffer Example**

```
class GfG
{
  public static void main(String args[])
  {
    int f = 1234;
    StringBuffer sb = new StringBuffer();
    sb.append(f);
    String str6 = sb.toString();
    System.out.println("String str6 = " + str6);
  }
}
```

or

```
class GfG
{
  public static void main(String args[])
  {
    String str6 = new StringBuffer().append(1234).toString();
    System.out.println("String str6 = " + str6);
  }
}
```

Output:

```
String str6 = 1234
```

**StringBuilder Example**

```
class GfG
{
  public static void main(String args[])
  {
    int g = 1234;
    StringBuilder sb = new StringBuilder();
    sb.append(g);
    String str7 = sb.toString();
    System.out.println("String str7 = " + str7);
  }
}
```

or

```
class GfG
```

```
{
  public static void main(String args[])
  {
    String str7 = new StringBuilder().append(1234).toString();
    System.out.println("String str7 = " + str7);
  }
}
```

Output:

```
String str7 = 1234
```

6. **Convert with special radix**
All of the examples above use the base (radix) 10. Follwing are convenient methods to convert to binary, octal, and hexadecimal system. Arbitrary custom number system is also supported.
**Examples :**
**Binary**

```
class GfG
{
  public static void main(String args[])
  {
    int h = 255;
    String binaryString = Integer.toBinaryString(h);
    System.out.println(binaryString);
  }
}
```

Output:

```
11111111
```

11111111 is the binary representation of the number 255.

**Ocatal**

```
class GfG
{
  public static void main(String args[])
  {
    int i = 255;
    String octalString = Integer.toOctalString(i);
    System.out.println(octalString);
  }
}
```

Output:

```
377
```

377 is the octal representation of the number 255.

**Hexadecimal**

```
class GfG
{
  public static void main(String args[])
  {
    int j = 255;
    String hexString = Integer.toHexString(j);
    System.out.println(hexString);
  }
}
```

Output:

```
ff
```

ff is the hexadecimal representation of the number 255.

**Custom Base/Radix**
you can use any other custom base/radix when converting an int to String.
Following example uses the base 7 number system.

```
class GfG
{
  public static void main(String args[])
  {
    int k = 255;
    String customString = Integer.toString(k, 7);
    System.out.println(customString);
  }
}
```

Output:

```
513
```

513 is the representation of the number 255 when written in the base 7 system.

This article is contributed by **Amit Khandelwal** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java

## String vs StringBuilder vs StringBuffer in Java

Consider below code with three concatenation functions with three different types of parameters, String, StringBuffer and StringBuilder.

```
// Java program to demonstrate difference between String,
// StringBuilder and StringBuffer
```

```
class Geeksforgeeks
{
    // Concatenates to String
    public static void concat1(String s1)
    {
        s1 = s1 + "forgeeks";
    }

    // Concatenates to StringBuilder
    public static void concat2(StringBuilder s2)
    {
        s2.append("forgeeks");
    }

    // Concatenates to StringBuffer
    public static void concat3(StringBuffer s3)
    {
        s3.append("forgeeks");
    }

    public static void main(String[] args)
    {
        String s1 = "Geeks";
        concat1(s1);  // s1 is not changed
        System.out.println("String: " + s1);

        StringBuilder s2 = new StringBuilder("Geeks");
        concat2(s2); // s2 is changed
        System.out.println("StringBuilder: " + s2);

        StringBuffer s3 = new StringBuffer("Geeks");
        concat3(s3); // s3 is changed
        System.out.println("StringBuffer: " + s3);
    }
}
```

**Output:**

```
String: Geeks
StringBuilder: Geeksforgeeks
StringBuffer: Geeksforgeeks
```

**Explanation:**
***1. Concat1*** : In this method, we pass a string “Geeks” and perform “s1 = s1 + ”forgeeks”. The string passed from main() is not changed, this is due to the fact that String is **immutable**. Altering the value of string creates another object and s1 in concat1() stores reference of new string. References s1 in main() and cocat1() refer to different strings.

***2. Concat2 :*** In this method, we pass a string “Geeks” and perform “s2.append(“forgeeks”)” which changes the actual value of the string (in main) to “Geeksforgeeks”. This is due to the simple fact that StringBuilder is **mutable** and hence changes its value.

***2. Concat3 :*** StringBuffer is similar to StringBuilder except one difference that StringBuffer is thread safe, i.e., multiple threads can use it without any issue. The thread safety brings a penalty of performance.

**Conclusion:**

- Objects of String are immutable, and objects of StringBuffer and StringBuilder are mutable.
- StringBuffer and StringBuilder are similar, but StringBuilder is faster and preferred over StringBuffer for single threaded program. If thread safety is needed, then StringBuffer is used.

This article is contributed by **Pranjal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

# Split() String method in Java with examples

The string split() method breaks a given string around matches of the given regular expression.

For Example:

```
Input String: 016-78967
Regular Expression: -
Output : {"016","78967"}
```

Following are the two variants of split() method in Java:

**1. Public String [ ] split ( String regex, int limit )**

```
Parameters:
regex - a delimiting regular expression
Limit - the result threshold

Returns:
An array of strings computed by splitting the given string.

Throws:
PatternSyntaxException - if the provided regular expression’s
                    syntax is invalid.
```

Limit parameter can have 3 values:

```
limit > 0 : If this is the case then the pattern will be
          applied at most limit-1 times, the resulting
          array’s length will not be more than n, and
          the resulting array’s last entry will contain
          all input beyond the last matched pattern.
limit
Here's how it works:


Let the string to be splitted be : geekss@for@geekss

Regex   Limit        Result
@          2     {“geekss”, ”for@geekss”}
@          5     {“geekss”, ”for”, ”geekss”}
@          -2      {“geekss”, ”for”, ”geekss”}
s         5      {“geek”, ”“, “@for@geek”, “”, “”}
s         -2     {“geek”, ” “, “@for@geek”, “”, “”}
```

```
s         0       {"geek", "", "@for@geek"}
```

Following are the Java example codes to demonstrate working of split()

**Example 1:**

```
// Java program to demonstrate working of split(regex,
// limit) with small limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("@", 2);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geekss
for@geekss
```

**Example 2:**

```
// Java program to demonstrate working of split(regex,
// limit) with high limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("@", 5);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geekss
for
geekss
```

**Example 3:**

```
// Java program to demonstrate working of split(regex,
// limit) with negative limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("@", -2);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geekss
for
geekss
```

**Example 4:**

```
// Java program to demonstrate working of split(regex,
// limit) with high limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("s", 5);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geek

@for@geek
```

**Example 5:**

```
// Java program to demonstrate working of split(regex,
// limit) with negative limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("s",-2);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geek

@for@geek
```

**Example 6:**

```
// Java program to demonstrate working of split(regex,
// limit) with 0 limit.
public class GFG
{
    public static void main(String args[])
    {
        String str = "geekss@for@geekss";
        String [] arrOfStr = str.split("s",0);

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
Geek

@for@geek
```

**2. public String[] split(String regex)**

This variant of split method takes a regular expression as parameter, and breaks the given string around matches of this regular expression regex. Here by default limit is 0.

```
Parameters:
regex - a delimiting regular expression


Returns:
An array of strings computed by splitting the given string.


Throws:
PatternSyntaxException - if the provided regular expression’s
                         syntax is invalid.
```

Here are some working example codes:

**Example 1:**

```
// Java program to demonstrate working of split()
public class GFG
{
    public static void main(String args[])
    {
        String str = "GeeksforGeeks:A Computer Science Portal";
        String [] arrOfStr = str.split(":");

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
GeeksforGeeks
A Computer Science Portal
```

**Example 2:**

```
// Java program to demonstrate working of split()
public class GFG
{
    public static void main(String args[])
    {
        String str = "GeeksforGeeksforStudents";
        String [] arrOfStr = str.split("for");

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
Geeks
Geeks
Students
```

It can be seen in the above example that the pattern/regular expression “for” is applied twice (because “for” is present two times in the string to be splitted)

**Example 3:**

```
// Java program to demonstrate working of split()
public class GFG
{
    public static void main(String args[])
    {
        String str = "Geeks for Geeks";
        String [] arrOfStr = str.split(" ");

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
Geeks
for
Geeks
```

**Example 4:**

```
// Java program to demonstrate working of split()
public class GFG
{
    public static void main(String args[])
    {
        String str = "Geekssss";
        String [] arrOfStr = str.split("s");

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
Geek
```

In the above example that trailing empty strings are not included in the resulting array arrOfStr.

**Example 5:**

```
// Java program to demonstrate working of split()
public class GFG
{
    public static void main(String args[])
    {
        String str = "GeeksforforGeeksfor   ";
        String [] arrOfStr = str.split("for");

        for (String a : arrOfStr)
            System.out.println(a);
    }
}
```

Output:

```
geeks

geeks
```

In the above example, the trailing spaces (hence not empty string) in the end becomes a string in the resulting array arrOfStr.

This article is contributed by **Vaibhav Bajpai**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your artic

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner**
**Company Wise Coding Practice**

Java

# Swap two Strings without using third user defined variable in Java

Given two string variables a and b, swap these variables without using temporary or third variable in Java. Use of library methods is allowed.

Examples:

```
Input: a = "Hello"
       b = "World"

Output:
Strings before swap: a = Hello and b = World
Strings after swap: a = World and b = Hello
```

The idea is to string concatenation and substring() method to perform this operation. The substring() method comes in two forms as listed below:

- **substring(int beginindex):** This fuction will return substring of the calling string starting from the index passed as argument to this function till the last character in the calling string.
- **substring(int beginindex,int endindex):** This function will return Substring of the calling string starting from the beginindex(inclusive) and ending at the endindex(exclusive) passed as argument to this function.

**Algorithm:**

```
1) Append second string to first string and
  store in first string:
  a = a + b

2) call the method substring(int beginindex, int endindex)
  by passing beginindex as 0 and endindex as,
     a.length() - b.length():
  b = substring(0,a.length()-b.length());

3) call the method substring(int beginindex) by passing
  b.length() as argument to store the value of initial
  b string in a
  a = substring(b.length());
```

```java
// Java program to swap two strings without using a temporary
// variable.
import java.util.*;

class Swap
{
    public static void main(String args[])
    {
        // Declare two strings
        String a = "Hello";
        String b = "World";

        // Print String before swapping
        System.out.println("Strings before swap: a = " +
                                    a + " and b = "+b);

        // append 2nd string to 1st
        a = a + b;

        // store intial string a in string b
        b = a.substring(0,a.length()-b.length());

        // store initial string b in string a
        a = a.substring(b.length());

        // print String after swapping
        System.out.println("Strings after swap: a = " +
                                   a + " and b = " + b);
    }
}
```

Output:

```
Strings before swap: a = Hello and b = World
Strings after swap: a = World and b = Hello
```

**References:**

http://docs.oracle.com/

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# String to Integer in Java – parseInt()

While operating upon strings, there are times when we need to convert a number represented as a string into an integer type. The method generally used is parseInt().

**How to use parseInt() method in Java?**

There are two variants of this method:

```
public static int parseInt(String s) throws NumberFormatException

  - This function parses the string argument as a signed
    decimal integer.
```

```
public static int parseInt(String s, int radix) throws NumberFormatException

  - This function parses the string argument as a signed
    integer in the radix specified by the second argument.
```

**Returns:**

In simple words, both the methods convert the string into its integer equivalent. The only difference being is that of the parameter radix. The first method can be considered as an equivalent of the second method with radix = 10 (Decimal).

**Parameters:**

- **s** – A string which needs to be converted to the integer. It can also have the first character as a minus sign '-' ('\u002D') or plus sign '+' ('\u002B') to represent the sign of the number.
- **radix** – The radix used while the string is being parsed.
  Note: This parameter is only specific to the second variant of the method.

**Exceptions:**

NumberFormatException is thrown by this method if any of the following situations occurs:

For both the variants:

- String is null or of zero length
- The value represented by the string is not a value of type int
- Specifically for the parseInt(String s, int radix) variant of the function:
  - The second argument radix is either smaller than Character.MIN_RADIX or larger than Character.MAX_RADIX
  - Any character of the string is not a digit of the specified radix, except that the first character may be a minus sign '-' ('\u002D') or plus sign '+' ('\u002B') provided that the string is longer than length 1

**Examples:**

```
parseInt("20") returns 20
parseInt("+20") returns 20
parseInt("-20") returns -20
parseInt("20", 16) returns 16, (2)*16^1 + (0)*16^0 = 32
parsparseInt("2147483648", 10) throws a NumberFormatException
parseInt("99", 8) throws a NumberFormatException
              as 9 is not an accepted digit of octal number system
parseInt("geeks", 28) throws a NumberFormatException
parseInt("geeks", 29) returns 11670324,
              Number system with base 29 can have digits 0-9
              followed by characters a,b,c... upto s
parseInt("geeksforgeeks", 29) throws a NumberFormatException as the
                       result is not an integer.
```

Click the Run on IDE button and try converting different strings to integer yourself:

```
// Java program to demonstrate working parseInt()
public class GFG
{
    public static void main(String args[])
    {
        int decimalExample = Integer.parseInt("20");
        int signedPositiveExample = Integer.parseInt("+20");
        int signedNegativeExample = Integer.parseInt("-20");
        int radixExample = Integer.parseInt("20",16);
        int stringExample = Integer.parseInt("geeks",29);

        // Uncomment the following code to check
        // NumberFormatException

        //  String invalidArguments = "";
        //  int emptyString = Integer.parseInt(invalidArguments);
        //  int outOfRangeOfInteger = Integer.parseInt("geeksforgeeks",29);
        //  int domainOfNumberSystem = Integer.parseInt("geeks",28);

        System.out.println(decimalExample);
        System.out.println(signedPositiveExample);
        System.out.println(signedNegativeExample);
        System.out.println(radixExample);
        System.out.println(stringExample);
    }
}
```

Output:

```
20
20
-20
```

```
32
11670324
```

Similarly, we can convert the string to any other primitive data types:

**parseLong()** – parses the string to Long

**parseDouble()** – parses the string to Double

If we want to convert the string to integer **without using parseInt()**, we can use **valueOf()** method. It also has two variants similar to parseInt().

**Difference between parseInt() and valueOf():**
parseInt() parses the string and returns the primitive integer type. However, valueOf() returns an Integer object.
**Note:** valueOf() uses parseInt() internally to convert to integer.

```
// Java program to demonstrate working of valueOf()
public class GFG
{
    public static void main(String args[])
    {
        int decimalExample = Integer.valueOf("20");
        int signedPositiveExample = Integer.valueOf("+20");
        int signedNegativeExample = Integer.valueOf("-20");
        int radixExample = Integer.valueOf("20",16);
        int stringExample = Integer.valueOf("geeks",29);

        System.out.println(decimalExample);
        System.out.println(signedPositiveExample);
        System.out.println(signedNegativeExample);
        System.out.println(radixExample);
        System.out.println(stringExample);
    }
}
```

Output :

```
20
20
-20
32
11670324
```

Related Article:
Java.lang.Integer class and its methods

References:
Offical Java Documentation

This article is contributed by **Shikhar Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## StringTokenizer class in Java with example | Set 1 ( Constructors)

StringTokenizer class in Java is used to break a string into tokens.

**Example:**
stringtokenizer

A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed.
A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

**Constructors:**

**StringTokenizer(String str) :**
**str** is string to be tokenized.
Considers default delimiters like new line, space, tab,
carriage return and form feed.

**StringTokenizer(String str, String delim) :**
**delim** is set of delimiters that are used to tokenize
the given string.

**StringTokenizer(String str, String delim, boolean flag):**
The first two parameters have same meaning. The flag
serves following purpose.

If the **flag** is **false**, delimiter characters serve to
separate tokens. For example, if string is "hello geeks"
and delimiter is " ", then tokens are "hello" and "geeks".

If the **flag** is **true**, delimiter characters are
considered to be tokens. For example, if string is "hello
geeks" and delimiter is " ", then tokens are "hello", " "
and "geeks".

```
/* A Java program to illustrate working of StringTokenizer
```

```
   class:*/
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        System.out.println("Using Constructor 1 - ");
        StringTokenizer st1 =
             new StringTokenizer("Hello Geeks How are you", " ");
        while (st1.hasMoreTokens())
            System.out.println(st1.nextToken());

        System.out.println("Using Constructor 2 - ");
        StringTokenizer st2 =
             new StringTokenizer("JAVA : Code : String", " :");
        while (st2.hasMoreTokens())
            System.out.println(st2.nextToken());

        System.out.println("Using Constructor 3 - ");
        StringTokenizer st3 =
             new StringTokenizer("JAVA : Code : String", " :",  true);
        while (st3.hasMoreTokens())
            System.out.println(st3.nextToken());
    }
}
```

**Output :**

```
Using Constructor 1 -
Hello
Geeks
How
are
you
Using Constructor 2 -
JAVA
Code
String
Using Constructor 3 -
JAVA

:

Code

:

String
```

We will soon be discussing methods of StringTokenizer in separate posts.

**Reference:**
https://docs.oracle.com/javase/7/docs/api/java/util/StringTokenizer.html

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Library

# Scanner Class in Java

Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double etc. and strings. It is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

- To create an object of Scanner class, we usually pass the predefined object System.in, which represents the standard input stream. We may pass an object of class File if we want to read input from a file.
- To read numerical values of a certain data type XYZ, the function to use is nextXYZ(). For example, to read a value of type short, we can use nextShort()
- To read strings, we use nextLine().
- To read a single character, we use next().charAt(0). next() function returns the next token/word in the input as a string and charAt(0) funtion returns the first character in that string.

Let us look at the code snippet to read data of various data types.

```
// Java program to read data of various types using Scanner class.
import java.util.Scanner;
public class ScannerDemo1
{
    public static void main(String[] args)
    {
        // Declare the object and initialize with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // String input
        String name = sc.nextLine();

        // Character input
        char gender = sc.next().charAt(0);

        // Numerical data input
        // byte, short and float can be read
        // using similar-named functions.
        int age = sc.nextInt();
        long mobileNo = sc.nextLong();
        double cgpa = sc.nextDouble();

        // Print the values to check if input was correctly obtained.
        System.out.println("Name: "+name);
        System.out.println("Gender: "+gender);
        System.out.println("Age: "+age);
        System.out.println("Mobile Number: "+mobileNo);
        System.out.println("CGPA: "+cgpa);
    }
}
```

Input :

```
Geek
F
40
9876543210
9.9
```

Output :

```
Name: Geek
Gender: F
Age: 40
Mobile Number: 9876543210
CGPA: 9.9
```

Sometimes, we have to check if the next value we read is of a certain type or if the input has ended (EOF marker encountered).

Then, we check if the scanner's input is of the type we want with the help of hasNextXYZ() functions where XYZ is the type we are interested in. The function returns true if the scanner has a token of that type, otherwise false. For example, in the above code, we have used hasNextInt().To check for a string, we use hasNextLine(). Similarly, to check for a single character, we use hasNext().charAt(0).

Let us look at the code snippet to read some numbers from console and print their mean.

```
// Java program to read some values using Scanner
// class and print their mean.
import java.util.Scanner;

public class ScannerDemo2
{
    public static void main(String[] args)
    {
        // Declare an object and initialize with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // Initialize sum and count of input elements
        int sum = 0, count = 0;

        // Check if an int value is available
        while (sc.hasNextInt())
        {
            // Read an int value
            int num = sc.nextInt();
            sum += num;
            count++;
        }
        int mean = sum / count;
        System.out.println("Mean: " + mean);
    }
}
```

Input:

```
101
223
238
892
99
500
728
```

Output:

```
Mean: 397
```

This article is contributed by **Sukrit Bhatnagar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java-I/O

# Scanner and nextChar() in Java

Scanner class in Java supports nextInt(), nextLong(), nextDouble() etc. But there is no nextChar() (See this for examples)

To read a char, we use **next().charAt(0)**. next() function returns the next token/word in the input as a string and charAt(0) function returns the first character in that string.

```
// Java program to read character using Scanner
// class
import java.util.Scanner;
public class ScannerDemo1
{
    public static void main(String[] args)
    {
        // Declare the object and initialize with
        // predefined standard input object
        Scanner sc = new Scanner(System.in);

        // Character input
        char c = sc.next().charAt(0);

        // Print the read value
        System.out.println("c = "+c);
    }
}
```

Input :

```
g
```

Output :

```
c = g
```

This article is contributed by **Piyush Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the

GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# Difference between Scanner and BufferReader Class in Java

java.util.Scanner class is a simple text scanner which can parse primitive types and strings. It internally uses regular expressions to read different types.

Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of sequence of characters

Following are differences between above two.

**Issue with Scanner when nextLine() is used after nextXXX()**

Try to guess the output of following code :

```
// Code using Scanner Class
import java.util.Scanner;
class Differ
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        System.out.println("Enter an integer");
        int a = scn.nextInt();
        System.out.println("Enter a String");
        String b = scn.nextLine();
        System.out.printf("You have entered:- "
                + a + " " + "and name as " + b);
    }
}
```

Input:

```
50
Geek
```

Output:

```
Enter an integer
Enter a String
You have entered:- 50 and name as
```

Let us try the same using Buffer class and same Input

```
// Code using Buffer Class
import java.io.*;
class Differ
{
    public static void main(String args[])
                throws IOException
    {
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        System.out.println("Enter an integer");
        int a = Integer.parseInt(br.readLine());
        System.out.println("Enter a String");
        String b = br.readLine();
        System.out.printf("You have entered:- " + a +
                    " and name as " + b);
    }
}
```

Input:

```
50
Geek
```

Output:

```
Enter an integer
Enter a String
you have entered:- 50 and name as Geek
```

In Scanner class if we call nextLine() method after any one of the seven nextXXX() method then the nextLine() doesn't not read values from console and cursor will not come into console it will skip that step. The nextXXX() methods are nextInt(), nextFloat(), nextByte(), nextShort(), nextDouble(), nextLong(), next().

In BufferReader class there is no such type of problem. This problem occurs only for Scanner class, due to nextXXX() methods ignore newline character and nextLine() only reads newline character. If we use one more call of nextLine() method between nextXXX() and nextLine(), then this problem will not occur because nextLine() will consume the newline character. See this for the corrected program. This problem is same as scanf() followed by gets() in C/C++.

**Other differences:**

- BufferedReader is synchronous while Scanner is not. BufferedReader should be used if we are working with multiple threads.
- BufferedReader has significantly larger buffer memory than Scanner.
- The Scanner has a little buffer (1KB char buffer) as opposed to the BufferedReader (8KB byte buffer), but it's more than enough.
- BufferedReader is a bit faster as compared to scanner because scanner does parsing of input data and BufferedReader simply reads sequence of characters.

This article is contributed by **Pranav Adarsh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java-I/O

# Formatted output in Java

Sometimes in Competitive programming, it is essential to print the output in a given specified format. Most users are familiar with printf function in C. Let us see discuss how we can format the output in Java:

**Formatting output using System.out.printf()**

This is the easiest of all methods as this is similar to printf in C. Note that System.out.print() and System.out.println() take a single argument, but printf() may take multiple arguments.

```
// A Java program to demonstrate working of printf() in Java
class JavaFormatter1
{
  public static void main(String args[])
  {
    int x = 100;
    System.out.printf("Printing simple integer: x = %d\n", x);

    // this will print it upto 2 decimal places
    System.out.printf("Formatted with precison: PI = %.2f\n", Math.PI);

    float n = 5.2f;

    // automatically appends zero to the rightmost part of decimal
    System.out.printf("Formatted to specific width: n = %.4f\n", n);

    n = 2324435.3f;

    // here number is formatted from right margin and occupies a
    // width of 20 characters
    System.out.printf("Formatted to right margin: n = %20.4f\n", n);
  }
}
```

Output:

```
Without formattiing: PI = 3.141592653589793
Formatted to Give precison: PI = 3.14
Formatted to specific width: n = 5.2000
Formatted to right margin: n =         2324435.2500
```

System.out.format() is equivalent to printf() and can also be used.

**Formatting using DecimalFormat class:**

DecimalFormat is used to format decimal numbers.

```
// Java program to demonstrate working of DecimalFormat
import java.text.DecimalFormat;

class JavaFormatter2
{
  public static void main(String args[])
  {
    double num = 123.4567;

    // prints only numeric part of a floating number
    DecimalFormat ft = new DecimalFormat("####");
    System.out.println("Without fraction part: num = " + ft.format(num));

    // this will print it upto 2 decimal places
    ft = new DecimalFormat("#.##");
    System.out.println("Formatted to Give precison: num = " + ft.format(num));

    // automatically appends zero to the rightmost part of decimal
    // instead of #,we use digit 0
    ft = new DecimalFormat("#.000000");
    System.out.println("appended zeroes to right: num = " + ft.format(num));

    // automatically appends zero to the leftmost of decimal number
    // instead of #,we use digit 0
    ft = new DecimalFormat("00000.00");
    System.out.println("formatting Numeric part : num = "+ft.format(num));

    // formatting money in dollars
    double income = 23456.789;
    ft = new DecimalFormat("$###,###.##");
    System.out.println("your Formatted Dream Income : " + ft.format(income));
  }
}
```

Output:

```
Without fraction part: num = 123
Formatted to Give precison: num = 123.46
appended zeroes to right: num = 123.456700
formatting Numeric part : num = 00123.46
your Formatted Dream Income : $23,456.79
```

**Formatting dates and parsing using SimpleDateFormat class:**

This class is present in java.text package.

```
// Java program to demonstrate working of SimpleDateFormat
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;

class Formatter3
{
  public static void main(String args[]) throws ParseException
  {
    // Formatting as per given pattern in the argument
    SimpleDateFormat ft = new SimpleDateFormat("dd-MM-yyyy");
    String str = ft.format(new Date());
    System.out.println("Formatted Date : " + str);

    // parsing a given String
    str = "02/18/1995";
    ft = new SimpleDateFormat("MM/dd/yyyy");
    Date date = ft.parse(str);

    // this will print the date as per parsed string
    System.out.println("Parsed Date : " + date);
  }
}
```

Output:

```
Formatted Date : 16-10-2016
Parsed Date : Sat Feb 18 00:00:00 UTC 1995
```

**References:**
https://docs.oracle.com/javase/tutorial/essential/io/formatting.html
https://docs.oracle.com/javase/tutorial/java/data/numberformat.html
http://docs.oracle.com/javase/6/docs/api/java/text/SimpleDateFormat.html

This article is contributed by **Pankaj Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Competitive Programming
Java

# Fast I/O in Java in Competitive Programming

Using Java in competitive programming is not something many people would suggest just because of its slow input and output, and well indeed it is slow.

In this article, we have discussed some ways to get around the difficulty and change the verdict from TLE to (in most cases) AC.

For all the Programs below
**Input:**

```
7 3
1
51
966369
7
9
999996
11
```

**Output:**

```
4
```

1. **Scanner Class** – (easy, less typing, but not recommended very slow, refer this for reasons of slowness): In most of the cases we get TLE while using scanner class. It uses built-in nextInt(), nextLong(), nextDouble methods to read the desired object after initiating scanner object with input stream.(eg System.in). The following program many a times gets time limit exceeded verdict and therefore not of much use.

```
// Working program using Scanner
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Scanner;
public class Main
{
    public static void main(String[] args)
    {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        int k = s.nextInt();
        int count = 0;
        while (n-- > 0)
        {
            int x = s.nextInt();
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

2. **BufferedReader –** (fast, but not recommended as it requires lot of typing): The Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. With this method we will have to parse the value every time for desired type. Reading multiple words from single line adds to its complexity because of the use of Stringtokenizer and hence this is not recommended. This gets accepted with a running time of approx 0.89 s.but still as you can see it requires a lot of typing all together and therefore method 3 is recommended.

```
// Working program using BufferedReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.StringTokenizer;

public class Main
{
    public static void main(String[] args) throws IOException
    {

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

        StringTokenizer st = new StringTokenizer(br.readLine());
        int n = Integer.parseInt(st.nextToken());
        int k = Integer.parseInt(st.nextToken());
        int count = 0;
        while (n-- > 0)
        {
            int x = Integer.parseInt(br.readLine());
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

3. **Userdefined FastReader Class-** (which uses bufferedReader and StringTokenizer): This method uses the time advantage of BufferedReader and StringTokenizer and the advantage of user defined methods for less typing and therefore a faster input altogether. This gets accepted with a time of 1.23 s and this method is *very much recommended* as it is easy to remember and is fast enough to meet the needs of most of the question in competitive coding.

```
// Working program with FastReader
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;
import java.util.StringTokenizer;

public class Main
{
    static class FastReader
    {
        BufferedReader br;
        StringTokenizer st;

        public FastReader()
        {
            br = new BufferedReader(new
                     InputStreamReader(System.in));
        }

        String next()
        {
            while (st == null || !st.hasMoreElements())
            {
                try
                {
                    st = new StringTokenizer(br.readLine());
                }
                catch (IOException  e)
                {
                    e.printStackTrace();
                }
            }
            return st.nextToken();
        }

        int nextInt()
        {
            return Integer.parseInt(next());
        }

        long nextLong()
        {
            return Long.parseLong(next());
        }

        double nextDouble()
        {
            return Double.parseDouble(next());
        }

        String nextLine()
        {
            String str = "";
            try
            {
                str = br.readLine();
            }
            catch (IOException e)
            {
                e.printStackTrace();
            }
            return str;
        }
    }

    public static void main(String[] args)
    {
        FastReader s=new FastReader();
        int n = s.nextInt();
        int k = s.nextInt();
        int count = 0;
        while (n-- > 0)
        {
            int x = s.nextInt();
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

4. **Using Reader Class**: There is yet another fast way through the problem, I would say the fastest way but is not recommended since it requires very cumbersome methods in its implementation. It uses inputDataStream to read through the stream of data and uses read() method and nextInt() methods for taking inputs. This is by far the fastest ways of taking input but is difficult to remember and is cumbersome in its approach. Below is the sample program using this method.
   This gets accepted with a surprising time of just 0.28 s. Although this is ultra fast, it is clearly not an easy method to remember.

```
// Working program using Reader Class
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;
import java.util.StringTokenizer;

public class Main
{
    static class Reader
    {
        final private int BUFFER_SIZE = 1 << 16;
        private DataInputStream din;
        private byte[] buffer;
        private int bufferPointer, bytesRead;

        public Reader()
        {
            din = new DataInputStream(System.in);
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public Reader(String file_name) throws IOException
        {
            din = new DataInputStream(new FileInputStream(file_name));
            buffer = new byte[BUFFER_SIZE];
            bufferPointer = bytesRead = 0;
        }

        public String readLine() throws IOException
        {
            byte[] buf = new byte[64]; // line length
            int cnt = 0, c;
            while ((c = read()) != -1)
            {
                if (c == '\n')
                    break;
                buf[cnt++] = (byte) c;
            }
            return new String(buf, 0, cnt);
        }

        public int nextInt() throws IOException
        {
            int ret = 0;
            byte c = read();
            while (c <= ' ')
                c = read();
            boolean neg = (c == '-');
            if (neg)
                c = read();
            do
            {
                ret = ret * 10 + c - '0';
            }  while ((c = read()) >= '0' && c <= '9');

            if (neg)
                return -ret;
            return ret;
        }

        public long nextLong() throws IOException
        {
            long ret = 0;
            byte c = read();
            while (c <= ' ')
                c = read();
            boolean neg = (c == '-');
            if (neg)
                c = read();
            do {
                ret = ret * 10 + c - '0';
            }
            while ((c = read()) >= '0' && c <= '9');
            if (neg)
                return -ret;
            return ret;
        }

        public double nextDouble() throws IOException
        {
            double ret = 0, div = 1;
            byte c = read();
            while (c <= ' ')
                c = read();
            boolean neg = (c == '-');
            if (neg)
                c = read();

            do {
                ret = ret * 10 + c - '0';
            }
            while ((c = read()) >= '0' && c <= '9');

            if (c == '.')
            {
                while ((c = read()) >= '0' && c <= '9')
                {
                    ret += (c - '0') / (div *= 10);
                }
            }

            if (neg)
                return -ret;
            return ret;
        }

        private void fillBuffer() throws IOException
        {
            bytesRead = din.read(buffer, bufferPointer = 0, BUFFER_SIZE);
            if (bytesRead == -1)
                buffer[0] = -1;
        }

        private byte read() throws IOException
        {
            if (bufferPointer == bytesRead)
                fillBuffer();
            return buffer[bufferPointer++];
        }

        public void close() throws IOException
        {
            if (din == null)
                return;
            din.close();
        }
    }

    public static void main(String[] args) throws IOException
    {
        Reader s=new Reader();
        int n = s.nextInt();
        int k = s.nextInt();
        int count=0;
        while (n-- > 0)
        {
            int x = s.nextInt();
            if (x%k == 0)
               count++;
        }
        System.out.println(count);
    }
}
```

Suggested Read: Enormous Input Test designed to check the fast input handling of your language.Timings of all programs are noted from SPOJ.

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Competitive Programming
Java

# Ways to read input from console in Java

In Java, there are three different ways for reading input from the user in the command line environment(console).

**1.Using Buffered Reader Class**

This is the Java classical method to take input, Introduced in JDK1.0. This method is used by wrapping the System.in (standard input stream) in an InputStreamReader which is wrapped in a BufferedReader, we can read input from the user in the command line.

**Advantages**

- The input is buffered for efficient reading.

**Drawback:**

- The wrapping code is hard to remember.

**Program:**

```
// Java program to demonstrate BufferedReader
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class Test
{
    public static void main(String[] args) throws IOException
    {
        //Enter data using BufferReader
        BufferedReader reader =
                   new BufferedReader(new InputStreamReader(System.in));

        // Reading data using readLine
        String name = reader.readLine();

        // Printing the read line
        System.out.println(name);
    }
}
```

Input:

```
Geek
```

Output:

```
Geek
```

Note: To read other types, we use functions like Integer.parseInt(), Double.parseDouble(). To read multiple values, we use split().

**2. Using Scanner Class**

This is probably the most preferred method to take input. The main purpose of the Scanner class is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line.

Advantages:

- Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
- Regular expressions can be used to find tokens.

**Drawback:**

- The reading methods are not synchronized

To see more differences, please see this article.

```
// Java program to demonstrate working of Scanner in Java
import java.util.Scanner;

class GetInputFromUser
{
    public static void main(String args[])
    {
        // Using Scanner for Getting Input from User
        Scanner in = new Scanner(System.in);

        String s = in.nextLine();
        System.out.println("You entered string "+s);

        int a = in.nextInt();
        System.out.println("You entered integer "+a);

        float b = in.nextFloat();
        System.out.println("You entered float "+b);
    }
}
```

Input:

```
GeeksforGeeks
```

```
12
3.4
```

Output:

```
You entered integer 12
Enter a float
You entered float 3.4
```

**3. Using Console Class**

It has been becoming a preferred way for reading user's input from the command line. In addition, it can be used for reading password-like input without echoing the characters entered by the user; the format string syntax can also be used (like System.out.printf()).
**Advantages:**

- Reading password without echoing the entered characters.
- Reading methods are synchronized.
- Format string syntax can be used.

**Drawback:**

- Does not work in non-interactive environment (such as in an IDE).

```
// Java program to demonstrate working of System.console()
// Note that this program does not work on IDEs as
// System.console() may require console
public class Sample
{
    public static void main(String[] args)
    {
        // Using Console to input data from user
        String name = System.console().readLine();

        System.out.println(name);
    }
}
```

Please refer this for more faster ways of reading input.

This article is contributed by **D Raj Ranu**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Technical Scripter
Java-I/O

# Arrays in Java

Unlike C++, arrays are first class objects in Java. For example, in the following program, size of array is accessed using *length* which is a member of *arr[]* object.

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
       int arr[] = {10, 20, 30, 40, 50};
       for(int i=0; i < arr.length; i++)
       {
             System.out.print(" " + arr[i]);
       }
    }
}
```

Output:
*10 20 30 40 50*

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# Default array values in Java

If we don't assign values to array elements, and try to access them, compiler does not produce error as in case of simple variables. Instead it assigns values which aren't garbage.

Below are the default assigned values.

- boolean : false
- int : 0
- double : 0.0
- String : null
- User Defined Type : null

```
// Java program to demonstrate default values of array
// elements
class ArrayDemo
{
    public static void main(String[] args)
    {
        System.out.println("String array default values:");
        String str[] = new String[5];
        for (String s : str)
            System.out.print(s + " ");

        System.out.println("\n\nInteger array default values:");
        int num[] = new int[5];
        for (int val : num)
             System.out.print(val + " ");

        System.out.println("\n\nDouble array default values:");
        double dnum[] = new double[5];
        for (double val : dnum)
            System.out.print(val + " ");

        System.out.println("\n\nBoolean array default values:");
        boolean bnum[] = new boolean[5];
        for (boolean val : bnum)
            System.out.print(val + " ");

        System.out.println("\n\nReference Array default values:");
        ArrayDemo ademo[] = new ArrayDemo[5];
        for (ArrayDemo val : ademo)
            System.out.print(val + " ");
    }
}
```

Output:

```
String array default values:
null null null null null

Integer array default values:
0 0 0 0 0

Double array default values:
0.0 0.0 0.0 0.0 0.0

Boolean array default values:
false false false false false

Reference Array default values:
null null null null null
```

This article is contributed by **Twinkle Tyagi** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# How to compare two arrays in Java?

Predict the output of following Java program.

```
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (arr1 == arr2) // Same as arr1.equals(arr2)
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Not Same
```

In Java, arrays are first class objects. In the above program, arr1 and arr2 are two references to two different objects. So when we compare arr1 and arr2, two reference variables are compared, therefore we get the output as “Not Same” (See this for more examples).

**How to compare array contents?**

A simple way is to run a loop and compare elements one by one. Java provides a direct method *Arrays.equals()* to compare two arrays. Actually, there is a list of equals() methods in Arrays class for different primitive types (int, char, ..etc) and one for Object type (which is base of all classes in Java).

```
// we need to import java.util.Arrays to use Arrays.equals().
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (Arrays.equals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Same
```

**How to Deep compare array contents?**

As seen above, the Arrays.equals() works fine and compares arrays contents. Now the questions, what if the arrays contain arrays inside them or some other references which refer to different object but have same values. For example, see the following program.

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        // inarr1 and inarr2 have same values
      int inarr1[] = {1, 2, 3};
      int inarr2[] = {1, 2, 3};
        Object[] arr1 = {inarr1};  // arr1 contains only one element
        Object[] arr2 = {inarr2};  // arr2 also contains only one element
        if (Arrays.equals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Not Same
```

So *Arrays.equals()* is not able to do deep comparison. Java provides another method for this Arrays.deepEquals() which does deep comparison.

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
     int inarr1[] = {1, 2, 3};
     int inarr2[] = {1, 2, 3};
        Object[] arr1 = {inarr1};  // arr1 contains only one element
        Object[] arr2 = {inarr2};  // arr2 also contains only one element
        if (Arrays.deepEquals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

Output:

```
Same
```

**How does Arrays.deepEquals() work?**
It compares two objects using any custom equals() methods they may have (if they have an equals() method implemented other than Object.equals()). If not, this method will then proceed to compare the objects field by field, recursively. As each field is encountered, it will attempt to use the derived equals() if it exists, otherwise it will continue to recurse further.
This method works on a cyclic Object graph like this: A->B->C->A. It has cycle detection so ANY two objects can be compared, and it will never enter into an endless loop (Source: https://code.google.com/p/deep-equals/).

**Exercise:** Predict the output of following program

```
import java.util.Arrays;
class Test
{
   public static void main (String[] args)
   {
      int inarr1[] = {1, 2, 3};
      int inarr2[] = {1, 2, 3};
      Object[] arr1 = {inarr1};  // arr1 contains only one element
      Object[] arr2 = {inarr2};  // arr2 also contains only one element
      Object[] outarr1 = {arr1}; // outarr1 contains only one element
      Object[] outarr2 = {arr2}; // outarr2 also contains only one element
      if (Arrays.deepEquals(outarr1, outarr2))
          System.out.println("Same");
      else
          System.out.println("Not same");
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Final arrays in Java

Predict the output of following Java program.

```
class Test
{
    public static void main(String args[])
    {
       final int arr[] = {1, 2, 3, 4, 5};  // Note: arr is final
       for (int i = 0; i < arr.length; i++)
       {
           arr[i] = arr[i]*10;
           System.out.println(arr[i]);
       }
    }
}
```

Output:

```
10
20
30
40
50
```

The array *arr* is declared as final, but the elements of array are changed without any problem. Arrays are objects and object variables are always references in Java. So, when we declare an object variable as final, it means that the variable cannot be changed to refer to anything else. For example, the following program 1 compiles without any error and program fails in compilation.

```
// Program 1
class Test
{
    int p = 20;
    public static void main(String args[])
    {
       final Test t = new Test();
       t.p = 30;
       System.out.println(t.p);
    }
}
```

Output: 30

```
// Program 2
class Test
```

```
{
   int p = 20;
   public static void main(String args[])
   {
      final Test t1 = new Test();
      Test t2 = new Test();
      t1 = t2;
      System.out.println(t1.p);
   }
}
```

Output: Compiler Error: cannot assign a value to final variable t1

*So a final array means that the array variable which is actually a reference to an object, cannot be changed to refer to anything else, but the members of array can be modified.*

As an exercise, predict the output of following program

```
class Test
{
   public static void main(String args[])
   {
      final int arr1[] = {1, 2, 3, 4, 5};
      int arr2[] = {10, 20, 30, 40, 50};
      arr2 = arr1;
      arr1 = arr2;
      for (int i = 0; i < arr2.length; i++)
         System.out.println(arr2[i]);
   }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# Jagged Array in Java

Prerequisite : Arrays in Java

Jagged array is array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D arrays but with variable number of columns in each row. These type of arrays are also known as Jagged arrays.

Following are Java programs to demonstrate the above concept.

```
// Program to demonstrate 2-D jagged array in Java
class Main
{
   public static void main(String[] args)
   {
      // Declaring 2-D array with 2 rows
      int arr[][] = new int[2][];

      // Making the above array Jagged

      // First row has 3 columns
      arr[0] = new int[3];

      // Second row has 2 columns
      arr[1] = new int[2];

      // Initializing array
      int count = 0;
      for (int i=0; i<arr.length; i++)
         for(int j=0; j<arr[i].length; j++)
            arr[i][j] = count++;

      // Displaying the values of 2D Jagged array
      System.out.println("Contents of 2D Jagged Array");
      for (int i=0; i<arr.length; i++)
      {
         for (int j=0; j<arr[i].length; j++)
            System.out.print(arr[i][j] + " ");
         System.out.println();
      }
   }
}
```

Output:

```
Contents of 2D Jagged Array
0 1 2
3 4
```

Following is another example where i'th row has i columns, i.e., first row has 1 element, second row has two elements and so on.

```
// Another Java program to demonstrate 2-D jagged
// array such that first row has 1 element, second
// row has two elements and so on.
class Main
{
   public static void main(String[] args)
   {
      int r = 5;

      // Declaring 2-D array with 5 rows
      int arr[][] = new int[r][];

      // Creating a 2D array such that first row
      // has 1 element, second row has two
      // elements and so on.
      for (int i=0; i<arr.length; i++)
         arr[i] = new int[i+1];

      // Initializing array
      int count = 0;
      for (int i=0; i<arr.length; i++)
         for(int j=0; j<arr[i].length; j++)
```

```
                arr[i][j] = count++;

        // Displaying the values of 2D Jagged array
        System.out.println("Contents of 2D Jagged Array");
        for (int i=0; i<arr.length; i++)
        {
            for (int j=0; j<arr[i].length; j++)
                System.out.print(arr[i][j] + " ");
            System.out.println();
        }
    }
}
```

Output:

```
Contents of 2D Jagged Array
0
1 2
3 4 5
6 7 8 9
10 11 12 13 14
```

This article is contributed by **Rahul Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Understanding Array IndexOutofbounds Exception in Java

Java supports creation and manipulation of arrays, as a data structure. The index of an array is an integer value that has value in interval [0, n-1], where n is the size of the array. If a request for a negative or an index greater than or equal to size of array is made, then the JAVA throws a ArrayIndexOutOfBounds Exception. This is unlike C/C++ where no index of bound check is done.

The ArrayIndexOutOfBoundsException is a Runtime Exception thrown only at runtime. The Java Compiler does not check for this error during the compilation of a program.

```
// A Common cause index out of bound
public class NewClass2
{
    public static void main(String[] args)
    {
        int ar[] = {1, 2, 3, 4, 5};
        for (int i=0; i<=ar.length; i++)
          System.out.println(ar[i]);
    }
}
```

**Runtime error throws an Exception:**

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
 at NewClass2.main(NewClass2.java:5)
```

**Output:**

```
1
2
3
4
5
```

Here if you carefully see, the array is of size 5. Therefore while accessing its element using for loop, the maximum value of index can be 4 but in our program it is going till 5 and thus the exception.

Let's see another example using arraylist:

```
// One more example with index out of bound
import java.util.ArrayList;
public class NewClass2
{
    public static void main(String[] args)
    {
        ArrayList<String> lis = new ArrayList<>();
        lis.add("My");
        lis.add("Name");
        System.out.println(lis.get(2));
    }
}
```

Runtime error here is a bit more informative than the previous time-

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 2, Size: 2
 at java.util.ArrayList.rangeCheck(ArrayList.java:653)
 at java.util.ArrayList.get(ArrayList.java:429)
 at NewClass2.main(NewClass2.java:7)
```

Lets understand it in a bit of detail-

- Index here defines the index we are trying to access.
- The size gives us information of the size of the list.
- Since size is 2, the last index we can access is (2-1)=1, and thus the exception

The correct way to access array is :

```
for (int i=0; i<ar.length; i++)
{
}
```

**Handling the Exception:**

- **Use for-each loop:** This automatically handles indices while accessing the elements of an array. Example-

  ```
  for(int m : ar){
  }
  ```

- **Use Try-Catch:** Consider enclosing your code inside a try-catch statement and manipulate the exception accordingly. As mentioned, Java won't let you access an invalid index and will definitely throw an ArrayIndexOutOfBoundsException. However, we should be careful inside the block of the catch statement, because if we don't handle the exception appropriately, we may conceal it and thus, create a bug in your application.

Quiz Question

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

# Array vs ArrayList in Java

In Java, following are two different ways to create an array.

1. **Array:** Simple fixed sized arrays that we create in Java, like below

```
int arr[] = new int[10]
```

2. **ArrayList :** Dynamic sized arrays in Java that implement List interface.

```
ArrayList<Type> arrL = new ArrayList<Type>();

Here Type is the type of elements in ArrayList to
be created
```

**Differences between Array and ArrayList**

- An array is basic functionality provided by Java. ArrayList is part of collection framework in Java. Therefore array members are accessed using [], while ArrayList has a set of methods to access elements and modify them.

```
// A Java program to demonstrate differences between array
// and ArrayList
import java.util.ArrayList;
import java.util.Arrays;

class Test
{
    public static void main(String args[])
    {
        /* ........... Normal Array............. */
        int[] arr = new int[3];
        arr[0] = 1;
        arr[1] = 2;
        System.out.println(arr[0]);

        /*............ArrayList..............*/
        // Create an arrayList with initial capacity 2
        ArrayList<Integer> arrL = new ArrayList<Integer>(2);

        // Add elements to ArrayList
        arrL.add(1);
        arrL.add(2);

        // Access elements of ArrayList
        System.out.println(arrL.get(0));
    }
}
```

Output:

```
1
1
```

.

- Array is a fixed size data structure while ArrayList is not. One need not to mention the size of Arraylist while creating its object. Even if we specify some initial capacity, we can add more elements.

```
// A Java program to demonstrate differences between array
// and ArrayList
import java.util.ArrayList;
import java.util.Arrays;
class Test
{
    public static void main(String args[])
    {
        /* ........... Normal Array............. */
        // Need to specify the size for array
        int[] arr = new int[3];
        arr[0] = 1;
        arr[1] = 2;
        arr[2] = 3;
        // We cannot add more elements to array arr[]

        /*............ArrayList..............*/
        // Need not to specify size
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);
        // We can add more elements to arrL

        System.out.println(arrL);
        System.out.println(Arrays.toString(arr));
    }
}
```

Output:

```
[1, 2, 3, 4]
[1, 2, 3]
```

.

- Array can contain both primitive data types as well as objects of a class depending on the definition of the array. However, ArrayList only supports object entries, not the primitive data types.
  Note: When we do arraylist.add(1); : it converts the primitive int data type into an Integer object. Sample Code:

```
import java.util.ArrayList;
class Test
{
    public static void main(String args[])
    {
       // allowed
        int[] array = new int[3];

        // allowed, however, need to be intialized
        Test[] array1 = new Test[3];

        // not allowed (Uncommenting below line causes
        // compiler error)
        // ArrayList<char> arrL = new ArrayList<char>();

        // Allowed
        ArrayList<Integer> arrL1 = new ArrayList<>();
        ArrayList<String> arrL2 = new ArrayList<>();
        ArrayList<object> arrL3 = new ArrayList<>();
    }
}
```

.

- Since ArrayList can't be created for primitive data types, members of ArrayList are always references to objects at different memory locations (See this for details). Therefore in ArrayList, the actual objects are never stored at contiguous locations. References of the actual objects are stored at contiguous locations.
  In array, it depends whether the arrays is of primitive type or object type. In case of primitive types, actual values are contiguous locations, but in case of objects, allocation is similar to ArrayList.

As a side note, ArrayList in Java can be seen as similar to vector in C++.

This article is contributed by **Pranjal Mathur**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

# ArrayList to Array Conversion in Java : toArray() Methods

Following methods can be used for converting ArrayList to Array:

arraylist-to-array

**Method 1: Using Object[] toArray() method**

**Syntax:**

```
public Object[] toArray()
```

- It is specified by toArray in interface Collection and interface List
- It overrides toArray in class AbstractCollection
- It returns an array containing all of the elements in this list in the correct order.

```
// Java program to demonstrate working of
// Objectp[] toArray()
import java.io.*;
import java.util.List;
import java.util.ArrayList;

class GFG
{
    public static void main (String[] args)
    {
        List<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);

        Object[] objects = al.toArray();

        // Printing array of objects
        for (Object obj : objects)
            System.out.print(obj + " ");
    }
}
```

Output:

```
10 20 30 40
```

Note: toArray() method returns an array of type Object(Object[]). We need to typecast it to Integer before using as Integer objects. If we do not typecast, we get compilation error. Consider the following example:

```
// A Java program to demonstrate that assigning Objects[]
// to Integer[] causes error.
import java.io.*;
import java.util.List;
import java.util.ArrayList;
```

```
class GFG
{
    public static void main (String[] args)
    {
        List<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);

        // Error: incompatible types: Object[]
        // cannot be converted to Integer[]
        Integer[] objects = al.toArray();

        for (Integer obj : objects)
            System.out.println(obj);
    }
}
```

Output:

```
19: error: incompatible types: Object[] cannot be converted to Integer[]
  Integer[] objects = al.toArray();
                                 ^
1 error
```

It is therefore recommended to create an array into which elements of List need to be stored and pass it as an argument in toArray() method to store elements if it is big enough. Otherwise a new array of the same type is allocated for this purpose.

**Method 2: Using T[] toArray(T[] a)**

```
// Converts a list into an array arr[] and returns same.
// If arr[] is not big enough, then a new array of same
// type is allocated for this purpose.
// T represents generic.
public  T[] toArray(T[] arr)
```

Note that the there is an array parameter and array return value. The main purpose of passed array is to tell the type of array. The returned array is of same type as passed array.

- If the passed array has enough space, then elements are stored in this array itself.
- If the passed array doesn't have enough space, a new array is created with same type and size of given list.
- If the passed array has more space, the array is first filled with list elements, then null values are filled.

It throws ArrayStoreException if the runtime type of a is not a supertype of the runtime type of every element in this list.

```
// A Java program to convert an ArrayList to arr[]
import java.io.*;
import java.util.List;
import java.util.ArrayList;

class GFG
{
    public static void main(String[] args)
    {
        List<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);

        Integer[] arr = new Integer[al.size()];
        arr = al.toArray(arr);

        for (Integer x : arr)
            System.out.print(x + " ");
    }
}
```

Output:

```
10 20 30 40
```

Note : If the specified array is null then it will throw NullpointerException. See this for example.

**Method 3: Manual method to covert ArrayList using get() method**

We can use this method if we don't want to use java in built toArray() method. This is a manual method of copying all the ArrayList elements to the String Array[].

```
// Returns the element at the specified index in the list.
public E get(int index)
```

```
// Java program to convert a ArrayList to an array
// using get() in a loop.
import java.io.*;
import java.util.List;
import java.util.ArrayList;

class GFG
{
    public static void main (String[] args)
    {
        List<Integer> al = new ArrayList<Integer>();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(40);

        Integer[] arr = new Integer[al.size()];

        // ArrayList to Array Conversion
        for (int i =0; i < al.size(); i++)
            arr[i] = al.get(i);

        for (Integer x : arr)
            System.out.print(x + " ");
    }
}
```

Output

```
10 20 30 40
```

**Reference:**

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/ArrayList.html#toArray()

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Custom ArrayList in Java

Prerequisite – ArrayList in Java

ArrayList in Java (equivalent to vector in C++) having dynamic size. It can be shrinked or expanded based on size. ArrayList is a part of collection framework and is present in java.util package.

An ArrayList:

```
ArrayList <E> list = new ArrayList <> ();
```

E here represents an object datatype e.g. **Integer**. The Integer class wraps a value of the primitive type **int** in an object. An object of type Integer contains a single field whose type is int.

More about Integer Object: here

list

**Custom ArrayList:** A custom arraylist has attributes based on user requirements and can have more than one type of data. This data is provided by a custom inner class which is formed by combination of various primitive object datatypes.

Consider a case when we have to take input as **N** number of students and details are:

**roll number, name, marks, phone number**

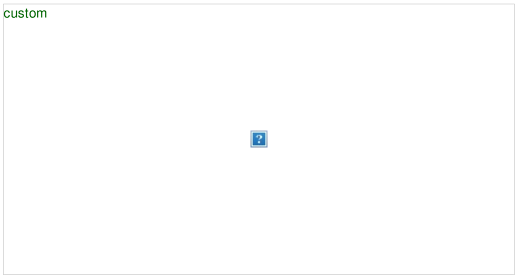
A normal method using object arraylist would be:

```
  // define 4 ArrayLists and save data accordingly in
  // each of them.

  // roll number arraylist
  ArrayList<Integer> roll = new ArrayList<>();

  // name arraylist
  ArrayList<String> name = new ArrayList<>();

  // marks arraylist
  ArrayList<Integer> marks = new ArrayList<>();

  // phone number arraylist
  ArrayList<Long> phone = new ArrayList<>();

  // and for n students
  for(int i = 0; i < n; i++)
  {
    // add all the values to each arraylist
    // each arraylist has primitive datatypes
    roll.add(rollnum_i);
    name.add(name_i);
    marks.add(marks_i);
    phone.add(phone_i);
  }
```

Now using a custom Arraylist:

The custom ArrayList simply supports multiple data in the way as shown in this image.

custom



To construct Custom ArrayList

- Build an ArrayList Object and place its type as a Class Data.
- Define a class and put the required entities in the constructor.
- Link those entities to global variables.
- Data received from the ArrayList is of that class type which stores multiple data.

```
// Java program to illustrate customArraylist in Java
import java.util.ArrayList;

class CustomArrayList
{
    // custom class which has data type
    // class has defined the type of data ArrayList
    // size of input 4
    int n=4;

    // the custom datatype class
    class Data
    {
        // global variables of the class
        int roll;
        String name;
        int marks;
        long phone;

        // constructor has type of data that is required
        Data(int roll, String name, int marks, long phone)
        {
            // initialize the input variable from main
            // function to the global variable of the class
            this.roll = roll;
            this.name = name;
            this.marks = marks;
            this.phone = phone;
        }
    }

    public static void main(String args[])
    {
        // suppose the custom input data
        int roll[] = {1, 2, 3, 4};
        String name[] = {"Shubham", "Atul", "Ayush", "Rupesh"};
        int marks[] = {100, 99, 93, 94};
        long phone[] = {8762357381L, 8762357382L, 8762357383L,
                        8762357384L
                       };

        // Create an object of the class
        CustomArrayList custom = new CustomArrayList();

        // and call the function to add the values to the arraylist
        custom.addValues(roll, name, marks, phone);
    }

    public void addValues(int roll[], String name[], int marks[],
                          long phone[])
    {
        // local custom arraylist of data type
        // Data having (int, String, int, long) type
        // from the class
        ArrayList<Data> list=new ArrayList<>();

        for (int i = 0; i < n; i++)
        {
            // create an object and send values to the
            // constructor to be saved in the Data class
            list.add(new Data(roll[i], name[i], marks[i],
                                              phone[i]));
        }

        // after adding values printing the values to test
        // the custom arraylist
        printValues(list);
    }

    public void printValues(ArrayList<Data> list)
    {
        // list- the custom arraylist is sent from
        // previous function

        for (int i = 0; i < n; i++)
        {
            // the data received from arraylist is of Data type
            // which is custom (int, String, int, long)
            // based on class Data

            Data data = list.get(i);

            // data variable of type Data has four primitive datatypes
            // roll -int
            // name- String
            // marks- int
            // phone- long
            System.out.println(data.roll+" "+data.name+" "
                             +data.marks+" "+data.phone);
        }
    }
}
```

Output:

```
1 Shubham 100 8762357381
2 Atul 99 8762357382
3 Ayush 93 8762357383
4 Rupesh 94 8762357384
```

**References:**

- Oracle documentation
- ArrayLists

This article is contributed by **Shubham Saxena**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Arrays class in Java

There are often times when we need to do following tasks on an array in Java.

- Fill an array with a particular value. We usually do it with the help of a for loop.
- Sort an array.

- Binary search in sorted array.
- And many more..

The Arrays class of the java.util package contains several static methods that we can use to fill, sort, search, etc in arrays. This class is a member of the Java Collections Framework and is present in **java.util.arrays**.

- **public static String toString(int[] a)** The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters a comma followed by a space. Elements are converted to strings as by String.valueOf(int). Returns "null" if a is null.

```
// Java program to demonstrate that we can print
// array elements in a single line
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {4, 6, 1, 8, 3, 9, 7, 4, 2};

        // To print the elements in one line
        System.out.println(Arrays.toString(ar));
    }
}
```

Output:

```
[4, 6, 1, 8, 3, 9, 7, 4, 2]
```

- **public static void sort(int[] a)** – Sorts the specified array into ascending numerical order.
- **public static void sort(int[] a, int fromIndex, int toIndex)** If we wish to sort a specified range of the array into ascending order. we can use this. The range to be sorted extends from the index fromIndex, inclusive, to the index toIndex, exclusive. If fromIndex == toIndex, the range to be sorted is empty.

```
// Java program to demonstrate that we can sort
// array elements in a single line
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {4, 6, 1, 8, 3, 9, 7, 4, 2};

        // To sort a specific range of array in
        // ascending order.
        Arrays.sort(ar, 0, 4);
        System.out.println("Sorted array in range" +
              " of 0-4 =>\n" + Arrays.toString(ar));

        // To sort the complete array in ascending order.
        Arrays.sort(ar);
        System.out.println("Completely sorted order =>\n"
                         + Arrays.toString(ar));
    }
}
```

Output:

```
Sorted array in range of 0-4 =>
[1, 4, 6, 8, 3, 9, 7, 4, 2]
Completely sorted order =>
[1, 2, 3, 4, 4, 6, 7, 8, 9]
```

- **public static int binarySearch(int[] a, int key)** Returns an int value for the index of the specified key in the specified array. Returns a negative number if the specified key is not found in the array. For this method to work properly, the array must first be sorted by the sort method.

```
// Java program to demonstrate that we can do
// binary search on array elements in a single line
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {4, 6, 1, 8, 3, 9, 7, 4, 2};

        // Sort the complete array in ascending order
        // so that Binary Search can be applied
        Arrays.sort(ar);

        // To search for a particular value(for eg 9)
        // use binarysearch method of arrays
        int index = Arrays.binarySearch(ar,9);
        System.out.println("Position of 9 in sorted"+
                   " arrays is => \n" + index);

    }
}
```

Output:

```
Position of 9 in sorted arrays is =>
8
```

- **public static int[] copyOf(int[] original, int newLength)** Copies the specified array and length. It truncates the array if provided length is smaller and pads if provided .
- **public static int[] copyOfRange(int[] original, int from, int to)** Copies the specified range of the specified array into a new array. The initial index of the range (from) must lie between zero and original.length, inclusive.

```
// Java program to demonstrate that we can copy
// an array or a subarray to a new array in single
// line.
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {4, 6, 1, 8, 3, 9, 7, 4, 2};

        // Copy the whole array
        int[] copy = Arrays.copyOf(ar, ar.length);
        System.out.println("Copied array => \n" +
                     Arrays.toString(copy));
        // Copy a specified range into a new array.
        int[] rcopy = Arrays.copyOfRange(ar, 1, 5);
        System.out.println("Copied subarray => \n" +
                           Arrays.toString(rcopy));
    }
}
```

Output:

```
Copied array =>
[4, 6, 1, 8, 3, 9, 7, 4, 2]
Copied subarray =>
[6, 1, 8, 3]
```

- **public static void fill(int[] a, int val)** Fills all elements of the specified array with the specified value.
- **public static void fill(int[] a, int fromIndex, int toIndex, int val)** – Fills elements of the specified array with the specified value from the fromIndex element, but not including the toIndex element.

```
// Java program to fill a subarray or complete
// array with given value.
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {4, 6, 1, 8, 3, 9, 7, 4, 2};

        // To fill a range with a particular value
        Arrays.fill(ar, 0, 3, 0);
        System.out.println("Array filled with 0 "+
          "from 0 to 3 => \n" + Arrays.toString(ar));

        // To fill complete array with a particular
        // value
        Arrays.fill(ar, 10);
        System.out.println("Array completely filled"+
                  " with 10=>\n"+Arrays.toString(ar));
    }
}
```

Output:

```
Array filled with 0 from 0 to 3 =>
[0, 0, 0, 8, 3, 9, 7, 4, 2]
Array completely filled with 10=>
[10, 10, 10, 10, 10, 10, 10, 10, 10]
```

- **public static List asList(T… a)** It takes an array and creates a wrapper that implements List, which makes the original array available as a list. Nothing is copied and all, only a single wrapper object is created. Operations on the list wrapper are propagated to the original array. This means that if you shuffle the list wrapper, the original array is shuffled as well, if you overwrite an element, it gets overwritten in the original array, etc. Of course, some List operations aren't allowed on the wrapper, like adding or removing elements from the list, you can only read or overwrite the elements. (Source Stackoverflow)

```
// Java program to demonstrate asList()
import java.util.Arrays;
import java.util.List;

public class Main
{
    public static void main(String[] args)
    {
        Integer ar[] = {4, 6, 1, 8, 3, 9, 7, 4, 2};

        // Creates a wrapper list over ar[]
        List<Integer> l1 = Arrays.asList(ar);

        System.out.println(l1);
    }
}
```

Output :

```
[4, 6, 1, 8, 3, 9, 7, 4, 2]
```

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# Arrays.deepToString() in Java with Example

java.util.Arrays.deepToString(Object[]) method is a java.util.Arrays class method.

Returns a string representation of the "deep contents" of the specified array. If the array contains other arrays as elements, the string representation contains their contents and so on. This method is designed for converting multidimensional arrays to strings. The simple toString() method works well for simple arrays, but doesn't work for multidimensional arrays. This method is designed for converting multi-dimensional arrays to strings.

**Syntax:**

```
public static String deepToString(Object[] arr)

arr - An array whose string representation is needed

This function returns string representation of arr[].
It returns "null"   if the specified array is null.
```

**Example:**

```
Let us suppose that we are making a 2-D array of
3 rows and 3 column.
   2  3  4
   5  6  7
   2  4  9

If use deepToString() method to print the 2-D array,
we will get string representation as :-
```

```
[[2,3,4], [5,6,7], [2,4,9]]
```

**Printing multidimensional Array**

```
// A Java program to print 2D array using deepToString()
import java.util.Arrays;

public class GfG
{
    public static void main(String[] args)
    {
        // Create a 2D array
        int[][] mat = new int[2][2];
        mat[0][0] = 99;
        mat[0][1] = 151;
        mat[1][0] = 30;
        mat[1][1] = 5;

        // print 2D integer array using deepToString()
        System.out.println(Arrays.deepToString(mat));
    }
}
```

Output:

```
[[99, 151], [30, 5]]
```

**toString() vs deepToString()**
toString() works well for single dimensional arrays, but doesn’t work for multidimensional arrays.

```
// Java program to demonstrate that toString works if we
// want to print single dimensional array, but doesn't work
// for multidimensional array.
import java.util.Arrays;
public class Deeptostring
{
    public static void main(String[] args)
    {
        // Trying to print array of strings using toString
        String[] strs = new String[] {"practice.geeksforgeeks.org",
                                      "quiz.geeksforgeeks.org"
                                     };
        System.out.println(Arrays.toString(strs));

        // Trying to print multidimensional array using
        // toString
        int[][] mat = new int[2][2];
        mat[0][0] = 99;
        mat[0][1] = 151;
        mat[1][0] = 30;
        mat[1][1] = 50;
        System.out.println(Arrays.toString(mat));
    }
}
```

Output:

```
[practice.geeksforgeeks.org, quiz.geeksforgeeks.org]
[[I@15db9742, [I@6d06d69c]
```

Note : We can use a loop to print contents of a multidimensional array using deepToString().

**deepToString() works for both single and multidimensional, but doesn’t work single dimensional array of primitives**

```
// Java program to demonstrate that deepToString(strs))
// works for single dimensional arrays also, but doesn't
// work single dimensional array of primitive types.
import java.util.Arrays;
public class Deeptostring
{
    public static void main(String[] args)
    {
        String[] strs = new String[] {"practice.geeksforgeeks.org",
                                      "quiz.geeksforgeeks.org"
                                     };
        System.out.println(Arrays.deepToString(strs));

        Integer []  arr1 = {10, 20, 30, 40};
        System.out.println(Arrays.deepToString(arr1));

        /* Uncommenting below code would cause error as
           deepToString() doesn't work for primitive types
        int []  arr2 = {10, 20, 30, 40};
        System.out.println(Arrays.deepToString(arr2));   */
    }
}
```

Output:

```
[practice.geeksforgeeks.org, quiz.geeksforgeeks.org]
[10, 20, 30, 40]
```

Reference:
https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#deepToString(java.lang.Object[])

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
.

## GATE CS Corner    Company Wise Coding Practice

Java

# Arrays.fill() in Java with Examples

**java.util.Arrays.fill()** method is in java.util.Arrays class. This method assigns the specified data type value to each element of the specified range of the specified array.

```
Syntax:
// Makes all elements of a[] equal to "val"
public static void fill(int[] a, int val)

// Makes elements from from_Index (inclusive) to to_Index
// (exclusive) equal to "val"
public static void fill(int[] a, int from_Index, int to_Index, int val)

This method doesn't return any value.
```

```
Exceptions it Throws:
IllegalArgumentException - if from_Index > to_Index
ArrayIndexOutOfBoundsException - if from_Index  a.length
```

**Examples:**

**We can fill entire array.**

```
// Java program to fill a subarray of given array
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {2, 2, 1, 8, 3, 2, 2, 4, 2};

        // To fill complete array with a particular
        // value
        Arrays.fill(ar, 10);
        System.out.println("Array completely filled" +
                  " with 10\n" + Arrays.toString(ar));
    }
}
```

Output:

```
Array completely filled with 10
[10, 10, 10, 10, 10, 10, 10, 10, 10]
```

**We can fill part of array.**

```
// Java program to fill a subarray array with
// given value.
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int ar[] = {2, 2, 2, 2, 2, 2, 2, 2, 2};

        // Fill from index 1 to index 4.
        Arrays.fill(ar, 1, 5, 10);

        System.out.println(Arrays.toString(ar));
    }
}
```

Output:

```
[2, 10, 10, 10, 10, 2, 2, 2, 2]
```

**We can fill a multidimensional array**
We can use a loop to fill a multidimensional array.

```
// Java program to fill a multidimensional array with
// given value.
import java.util.Arrays;

public class Main
{
    public static void main(String[] args)
    {
        int [][]ar = new int [3][4];

        // Fill each row with 10.
        for (int[] row : ar)
            Arrays.fill(row, 10);

        System.out.println(Arrays.deepToString(ar));
    }
}
```

Output:

```
[[10, 10, 10, 10], [10, 10, 10, 10], [10, 10, 10, 10]]
```

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Java

# Arrays.sort() in Java with examples

sort() method is a java.util.Arrays class method.

**Syntax:**

```
public static void sort(int[] arr, int from_Index, int to_Index)

arr - the array to be sorted
from_Index - the index of the first element, inclusive, to be sorted
to_Index - the index of the last element, exclusive, to be sorted
```

```
This method doesn't return any value.
```

A Java program to **sort an array of integers in ascending order**.

```
// A sample Java program to sort an array of integers
// using Arrays.sort(). It by default sorts in
// ascending order
import java.util.Arrays;

public class SortExample
{
    public static void main(String[] args)
    {
        // Our arr contains 8 elements
        int[] arr = {13, 7, 6, 45, 21, 9, 101, 102};

        Arrays.sort(arr);

        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Output:

```
Modified arr[] : [6, 7, 9, 13, 21, 45, 101, 102]
```

**We can also use sort() to sort a subarray of arr[]**

```
// A sample Java program to sort a subarray
// using Arrays.sort().
import java.util.Arrays;

public class SortExample
{
    public static void main(String[] args)
    {
        // Our arr contains 8 elements
        int[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

        // Sort subarray from index 1 to 5, i.e.,
        // only sort subarray {7, 6, 45, 21, 9} and
        // keep other elements as it is.
        Arrays.sort(arr, 1, 5);

        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Output:

```
Modified arr[] : [13, 6, 7, 21, 45, 9, 2, 100]
```

**We can also sort in descending order.**

```
// A sample Java program to sort a subarray
// in descending order using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
    public static void main(String[] args)
    {
        // Note that we have Integer here instead of
        // int[] as Collections.reverseOrder doesn't
        // work for primitive types.
        Integer[] arr = {13, 7, 6, 45, 21, 9, 2, 100};

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : %s",
                          Arrays.toString(arr));
    }
}
```

Output:

```
Modified arr[] : [100, 45, 21, 13, 9, 7, 6, 2]
```

**We can also sort strings in alphabetical order.**

```
// A sample Java program to sort an array of strings
// in ascending and descending orders using Arrays.sort().
import java.util.Arrays;
import java.util.Collections;

public class SortExample
{
    public static void main(String[] args)
    {
        String arr[] = {"practice.geeksforgeeks.org",
                        "quiz.geeksforgeeks.org",
                        "code.geeksforgeeks.org"
                       };

        // Sorts arr[] in ascending order
        Arrays.sort(arr);
        System.out.printf("Modified arr[] : \n%s\n\n",
                          Arrays.toString(arr));

        // Sorts arr[] in descending order
        Arrays.sort(arr, Collections.reverseOrder());

        System.out.printf("Modified arr[] : \n%s\n\n",
                          Arrays.toString(arr));
    }
}
```

Output:

```
Modified arr[] :
[code.geeksforgeeks.org, practice.geeksforgeeks.org, quiz.geeksforgeeks.org]

Modified arr[] :
[quiz.geeksforgeeks.org, practice.geeksforgeeks.org, code.geeksforgeeks.org]
```

**We can also sort an array according to user defined criteria.**

We use Comparator interface for this purpose. Below is an example.

```
// Java program to demonstrate working of Comparator
// interface
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                           String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                       " " + this.address;
    }
}

class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        Student [] arr = {new Student(111, "bbbb", "london"),
                          new Student(131, "aaaa", "nyc"),
                          new Student(121, "cccc", "jaipur")};

        System.out.println("Unsorted");
        for (int i=0; i<arr.length; i++)
            System.out.println(arr[i]);

        Arrays.sort(arr, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i=0; i<arr.length; i++)
            System.out.println(arr[i]);
    }
}
```

Output:

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc
```

**Arrays.sort() vs Collections.sort()**

Arrays.sort works for arrays which can be of primitive data type also. Collections.sort() works for objects Collections like ArrayList, LinkedList, etc.

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

# java.util.Arrays.copyof() in Java with examples

java.util.Arrays.copyof() method is in java.util.Arrays class. It copies the specified array, truncating or padding with false (if necessary) so the copy has the specified length.

**Syntax:**

```
copyOf(int[] original, int newLength)
```

- **original** – original array
- **newLength** – copy of original array

```
// Java program to illustrate copyof method
import java.util.Arrays;

public class Main
{
    public static void main(String args[])
    {
        // initializing an array original
        int[] org = new int[] {1, 2 ,3};

        System.out.println("Original Array");
        for (int i = 0; i < org.length; i++)
            System.out.print(org[i] + " ");

        // copying array org to copy
        int[] copy = Arrays.copyOf(org, 5);

        // Changing some elements of copy
        copy[3] = 11;
        copy[4] = 55;

        System.out.println("\nNew array copy after modifications:");
        for (int i = 0; i < copy.length; i++)
            System.out.print(copy[i] + " ");
    }
}
```

Output:

```
Original Array
1 2 3
New array copy after modifications:
1 2 3 11 55
```

**What happens if length of copied array is greater than original array?**
The two arrays will have same values for all the indices that are valid in original array and new array.
However, the indices missing in original will have **zero** in copy in case the copied array length is more than the original array.

```
// Java program to illustrate copyOf when new array
// is of higher length.
import java.util.Arrays;

public class Main
{
  public static void main(String args[])
  {
     // initializing an array original
     int[] org = new int[] {1, 2 ,3};

     System.out.println("Original Array : \n");
     for (int i = 0; i < org.length; i++)
         System.out.print(org[i] + " ");

      // copying array org to copy
      // Here, new array has 5 elements - two
      // elements more than the original array
      int[] copy = Arrays.copyOf(org, 5);

      System.out.print("\nNew array copy (of higher length):\n");
      for (int i = 0; i < copy.length; i++)
          System.out.print(copy[i] + " ");
    }
}
```

Output:

```
Original Array :
1 2 3
New array copy (of higher length):
1 2 3 0 0
```

If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Arrays.toString() in Java with Examples

Today we are going to discuss the simplest way to print the array as a string in Java: Arrays.toString() method.

**How to use Arrays.toString() method?**

**Description:**
Returns a string representation of the contents of the specified array. The string representation consists of a list of the array's elements, enclosed in square brackets ("[]"). Adjacent elements are separated by the characters ", " (a comma followed by a space). Returns "null" if a is null.

In case of an Object Array, if the array contains other arrays as elements, they are converted to strings by the Object.toString() method inherited from Object, which describes their identities rather than their contents.

**Variants:**

- public static String toString(boolean[] arr)
- public static String toString(byte[] arr)
- public static String toString(char[] arr)
- public static String toString(double[] arr)
- public static String toString(float[] arr)
- public static String toString(int[] arr)
- public static String toString(long[] arr)
- public static String toString(Object[] arr)
- public static String toString(short[] arr)

**Parameters:**
arr – the array whose string representation to return

**Returns:**

the string representation of arr

**Usage:**

The below mentioned Java code depicts the usage of the toString() method of Arrays class with examples:

```
// Java program to demonstrate working of Arrays.toString()
import java.io.*;
import java.util.*;

class GFG
{
    public static void main (String[] args)
    {
        // Let us create different types of arrays and
        // print their contents using Arrays.toString()
        boolean[] boolArr = new boolean [] {true,true,false,true};
        byte[] byteArr = new byte[] {10,20,30};
        char[] charArr = new char [] {'g','e','e','k','s'};
        double[] dblArr = new double [] {1,2,3,4};
        float[] floatArr = new float [] {1,2,3,4};
        int[] intArr = new int [] {1,2,3,4};
        long[] lomgArr = new long [] {1,2,3,4};
        Object[] objArr = new Object [] {1,2,3,4};
        short[] shortArr = new  short [] {1,2,3,4};

        System.out.println(Arrays.toString(boolArr));
        System.out.println(Arrays.toString(byteArr));
        System.out.println(Arrays.toString(charArr));
        System.out.println(Arrays.toString(dblArr));
        System.out.println(Arrays.toString(floatArr));
        System.out.println(Arrays.toString(intArr));
        System.out.println(Arrays.toString(lomgArr));
        System.out.println(Arrays.toString(objArr));
        System.out.println(Arrays.toString(shortArr));
    }
}
```

Output:

```
[true, true, false, true]
[10, 20, 30]
[g, e, e, k, s]
[1.0, 2.0, 3.0, 4.0]
[1.0, 2.0, 3.0, 4.0]
[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]
[1, 2, 3, 4]
```

**We can also use Arrays.toString() for objects of user defined class.**

Since Arrays.toString() is overloaded for array of Object class (there exist a method Arrays.toString(Object [])) and Object is ancestor of all classes, we can use call it for an array of any type of object.

```
// Java program to demonstrate working of Arrays.toString()
// for user defined objects.
import java.lang.*;
import java.util.*;

// Driver class
class Main
{
    public static void main (String[] args)
    {
        Student [] arr = {new Student(111, "bbbb", "london"),
                          new Student(131, "aaaa", "nyc"),
                          new Student(121, "cccc", "jaipur")};

        System.out.println(Arrays.toString(arr));
    }
}

// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                           String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                           " " + this.address;
    }
}
```

Output:

```
[111 bbbb london, 131 aaaa nyc, 121 cccc jaipur]
```

**Why does Object.toString() not work for Arrays?**

Using the toString() method on Arrays might not work. It considers an array as a typical object and returns default string, i.e., a '[' representing an array, followed by a character representing the primitive data type of array followed by an Identity Hex Code [See this for details]

This article is contributed by **Shikhar Goel**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

# Arrays.binarySearch() in Java with examples | Set 1

**Arrays.binarySearch()** is the simplest and most efficient method to find an element in a sorted array in Java

**Declaration:**

```
public static int binarySearch(data_type arr, data_type key )
```

where **data_type** can be any of the primitive data types: **byte**, **char**, **double**, **int**, **float**, **short**, **long** and **Object** as well.

**Description:**
Searches the specified array of the given data type for the specified value using the binary search algorithm. The array must be sorted (as by the Arrays.sort() method) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

**Parameters:**
arr – the array to be searched
key – the value to be searched for

**Returns:**
index of the search key, if it is contained in the array; otherwise, (-(insertion point) – 1). The insertion point is defined as the point at which the key would be inserted into the array: the index of the first element greater than the key, or a.length if all elements in the array are less than the specified key. Note that this guarantees that the return value will be >= 0 if and only if the key is found.

**Examples:**

```
Searching for 35 in byteArr[] = {10,20,15,22,35}
will give result as 4 as it is the index of 35

Searching for 35 in charArr[] = {'g','p','q','c','i'}
will give result as 1 as it is the index of 'g'

Searching for 22 in intArr[] = {10,20,15,22,35};
will give result as 3 as it is the index of 22

Searching for 1.5 in doubleArr[] = {10.2,15.1,2.2,3.5}
will give result as -1 as it is the insertion point of 1.5

Searching for 35.0 in floatArr[] = {10.2f,15.1f,2.2f,3.5f}
will give result as -5 as it is the insertion point of 35.0

Searching for 5 in shortArr[] = {10,20,15,22,35}
will give result as -1 as it is the insertion point of 5
```

```
// Java program to demonstrate working of Arrays.
// binarySearch() in a sorted array.
import java.util.Arrays;

public class GFG
{
    public static void main(String[] args)
    {
        byte byteArr[] = {10,20,15,22,35};
        char charArr[] = {'g','p','q','c','i'};
        int intArr[] = {10,20,15,22,35};
        double doubleArr[] = {10.2,15.1,2.2,3.5};
        float floatArr[] = {10.2f,15.1f,2.2f,3.5f};
        short shortArr[] = {10,20,15,22,35};

        Arrays.sort(byteArr);
        Arrays.sort(charArr);
        Arrays.sort(intArr);
        Arrays.sort(doubleArr);
        Arrays.sort(floatArr);
        Arrays.sort(shortArr);

        byte byteKey = 35;
        char charKey = 'g';
        int intKey = 22;
        double doubleKey = 1.5;
        float floatKey = 35;
        short shortKey = 5;

        System.out.println(byteKey + " found at index = "
                           +Arrays.binarySearch(byteArr,byteKey));
        System.out.println(charKey + " found at index = "
                           +Arrays.binarySearch(charArr,charKey));
        System.out.println(intKey + " found at index = "
                           +Arrays.binarySearch(intArr,intKey));
        System.out.println(doubleKey + " found at index = "
                           +Arrays.binarySearch(doubleArr,doubleKey));
        System.out.println(floatKey + " found at index = "
                           +Arrays.binarySearch(floatArr,floatKey));
        System.out.println(shortKey + " found at index = "
                           +Arrays.binarySearch(shortArr,shortKey));
    }
}
```

**Output:**

```
35 found at index = 4
g found at index = 1
22 found at index = 3
1.5 found at index = -1
35.0 found at index = -5
5 found at index = -1
```

**Important Points:**

- If input list is not sorted, the results are undefined.
- If there are duplicates, there is no guarantee which one will be found.

**Arrays.binarysearch() vs Collections.binarysearch()**
Arrays.binarysearch() works for arrays which can be of primitive data type also. Collections.binarysearch() works for objects Collections like ArrayList and LinkedList.

There are variants of this method in which we can also specify the range of array to search in. We will be discussing that as well as searching in an Object array in further posts.

This article is contributed by **Shikhar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Arrays.binarySearch() in Java with examples | Set 2 (Search in subarray)

Arrays.binarySearch()| Set 1 covers how to to find an element in a sorted array in Java. This set will cover "How to Search a key in an array within given range including only start index".

**Syntax :**

```
public static int binarySearch(data_type[] arr, int fromIndex, int toIndex, data_type key)
Parameters :
arr – the array to be searched
fromIndex – the index of the first element (inclusive) to be searched
toIndex - the index of the last element (exclusive) to be searched
 key  – the value to be searched for
```

- It is static inbuilt method defined in **Arrays (java.util.Arrays)** class in java and returns the index of the specified key if found within the specified range.
- Here, **data_type** can be any of the primitive **data_type**: byte, char, double, int, float, short, long and Object as well.
- The above function Searches a range of the specified array of the given data type for the specified key using binary search algorithm.
- The range within which the specified key to be searched must be sorted (as by the Arrays.sort() method) prior to making this call. Otherwise result would be undefined. If the specified array contains multiple values same as the specified key, there is no guarantee which one will be found.

**Returns :**
Index of the specified key if found within the specified range in the specified array, Otherwise (-(insertion point) – 1).
The insertion point is defined as a point at which the specified key would be inserted: the index of the first element in the range greater than the key, or toIndex if all elements in the range are less than the specified key.
Note:This guarantees that the return value will be >= 0 if and only if the key is found.

**Examples:**

```
 byteArr[] = {10,20,15,22,35}
 key = 22 to be searched between the range 2 to 4 in specified array.
 Output: 3

 charArr[] = {'g','p','q','c','i'}
 key = p to be searched between the range 1 to 4 in specified array.
 Output: 3

 intArr[] = {1,2,3,4,5,6}
 key = 3 to be searched between the range 1 to 4 in specified array.
 Output: 2

 doubleArr[] = {10.2,15.1,2.2,3.5}
 key = 1.5 to be searched between the range 1 to 4 in specified array.
 Output: -2 as it is the insertion point of 1.5

 floatArr[] = {10.2f,15.1f,2.2f,3.5f}
 key = 35.0 to be searched between the range 1 to 4 in specified array.
 Output: -5

shortArr[] = {10,20,15,22,35}
key = 5 to be searched between the range 0 to 4 in specified array.
Output: -1
```

**Implementation :**

```
// Java program to demonstrate working of  binarySearch() method
// for specified range in a sorted array.
import java.util.Arrays;

public class GFG
{
 public static void main(String[] args)
 {
  byte byteArr[] = {10,20,15,22,35};
  char charArr[] = {'g','p','q','c','i'};
  int intArr[] = {1,2,3,4,5,6};
  double doubleArr[] = {10.2,15.1,2.2,3.5};
  float floatArr[] = {10.2f,15.1f,2.2f,3.5f};
  short shortArr[] = {10,20,15,22,35};

  Arrays.sort(byteArr);
  Arrays.sort(charArr);
  Arrays.sort(intArr);
  Arrays.sort(doubleArr);
  Arrays.sort(floatArr);
  Arrays.sort(shortArr);

  byte byteKey = 22;
  char charKey = 'p';
  int intKey = 3;
  double doubleKey = 1.5;
  float floatKey = 35;
  short shortKey = 5;

  System.out.println(byteKey + " found at index = "
      +Arrays.binarySearch(byteArr,2,4,byteKey));
  System.out.println(charKey + " found at index = "
      +Arrays.binarySearch(charArr,1,4,charKey));
  System.out.println(intKey + " found at index = "
      +Arrays.binarySearch(intArr,1,4,intKey));
  System.out.println(doubleKey + " found at index = "
      +Arrays.binarySearch(doubleArr,1,4,doubleKey));
  System.out.println(floatKey + " found at index = "
      +Arrays.binarySearch(floatArr,1,4,floatKey));
  System.out.println(shortKey + " found at index = "
      +Arrays.binarySearch(shortArr,0,4,shortKey));
 }
}
```

Output:

```
22 found at index = 3
p found at index = 3
3 found at index = 2
1.5 found at index = -2
35.0 found at index = -5
5 found at index = -1
```

**Exceptions :**

1. **IllegalArgumentException** : throws when starting index(fromIndex) is greater than the end index(toIndex) of specified range.(means: fromIndex > toIndex)
2. **ArrayIndexOutOfBoundsException** : throws if one or both index are not valid means fromIndex<0 or toIndex > arr.length.

**Important Points:**

- If input list is not sorted, the results are undefined.
- If there are duplicates, there is no guarantee which one will be found.

**Reference :**
https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#binarySearch(int[],%20int)

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java
Java-Library

# Comparison of static keyword in C++ and Java

Static keyword is used for almost same purpose in both C++ and Java. There are some differences though. This post covers similarities and differences of static keyword in C++ and Java.

**Static Data Members:** Like C++, static data members in Java are class members and shared among all objects. For example, in the following Java program, static variable *count* is used to count the number of objects created.

```
class Test {
    static int count = 0;

    Test() {
       count++;
    }
    public static void main(String arr[]) {
       Test t1 = new Test();
       Test t2 = new Test();
       System.out.println("Total " + count + " objects created");
    }
}
```

Output:

```
Total 2 objects created
```

**Static Member Methods:** Like C++, methods declared as static are class members and have following restrictions:

**1)** They can only call other static methods. For example, the following program fails in compilation. fun() is non-static and it is called in static main()

```
class Main {
    public static void main(String args[]) {
        System.out.println(fun());
    }
    int fun() {
        return 20;
    }
}
```

**2)** They must only access static data.

**3)** They cannot access *this* or *super* . For example, the following program fails in compilation.

```
class Base {
    static int x = 0;
}

class Derived extends Base
{
   public static void fun() {
       System.out.println(super.x); // Compiler Error: non-static variable
                                 // cannot be referenced from a static context
   }
}
```

Like C++, static data members and static methods can be accessed without creating an object. They can be accessed using class name. For example, in the following program, static data member count and static method fun() are accessed without any object.

```
class Test {
    static int count = 0;
    public static void fun() {
       System.out.println("Static fun() called");
    }
}

class Main
{
    public static void main(String arr[]) {
       System.out.println("Test.count = " + Test.count);
       Test.fun();
    }
}
```

**Static Block:** Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initialization of a class. This code inside static block is executed only once. See Static blocks in Java for details.

**Static Local Variables:** Unlike C++, Java doesn't support static local variables. For example, the following Java program fails in compilation.

```
class Test {
   public static void main(String args[]) {
     System.out.println(fun());
   }
   static int fun()  {
     static int x= 10; //Compiler Error: Static local variables are not allowed
     return x--;
   }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Static blocks in Java

Unlike C++, Java supports a special block, called static block (also called static clause) which can be used for static initializations of a class. This code inside static block is executed only once: the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class). For example, check output of following Java program.

```
// filename: Main.java
class Test {
    static int i;
    int j;

    // start of static block
    static {
        i = 10;
        System.out.println("static block called ");
    }
    // end of static block
}

class Main {
    public static void main(String args[]) {

        // Although we don't have an object of Test, static block is
        // called because i is being accessed in following statement.
        System.out.println(Test.i);
    }
}
```

Output:

```
static block called
10
```

Also, static blocks are executed before constructors. For example, check output of following Java program.

```
// filename: Main.java
class Test {
    static int i;
    int j;
    static {
        i = 10;
        System.out.println("static block called ");
    }
    Test(){
        System.out.println("Constructor called");
    }
}

class Main {
    public static void main(String args[]) {

       // Although we have two objects, static block is executed only once.
       Test t1 = new Test();
       Test t2 = new Test();
    }
}
```

Output:

```
static block called
Constructor called
Constructor called
```

**What if we want to execute some code for every object?**

We use Initializer Block in Java

**References:**

Thinking in Java Book

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Are static local variables allowed in Java?

Unlike C/C++, static local variables are not allowed in Java. For example, following Java program fails in compilation with error *"Static local variables are not allowed"*

```
class Test {
   public static void main(String args[]) {
     System.out.println(fun());
   }

   static int fun()
   {
     static int x= 10;  //Error: Static local variables are not allowed
     return x--;
   }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Static class in Java

**Can a class be static in Java ?**

The answer is YES, we can have static class in java. In java, we have static instance variables as well as static methods and also static block. Classes can also be made static in Java.

Java allows us to define a class within another class. Such a class is called a nested class. The class which enclosed nested class is known as Outer class. In java, we can't make Top level class static. ***Only nested classes can be static***.

**What are the differences between static and non-static nested classes?**

Following are major differences between static nested class and non-static nested class. Non-static nested class is also called Inner Class.

**1)** Nested static class doesn't need reference of Outer class, but Non-static nested class or Inner class requires Outer class reference.

**2)** Inner class(or non-static nested class) can access both static and non-static members of Outer class. A static class cannot access non-static members of the Outer class. It can access only static members of Outer class.

**3)** An instance of Inner class cannot be created without an instance of outer class and an Inner class can reference data and methods defined in Outer class in which it nests, so we don't need to pass reference of an object to the constructor of the Inner class. For this reason Inner classes can make program simple and concise.

```
/* Java program to demonstrate how to implement static and non-static
   classes in a java program. */
class OuterClass{
   private static String msg = "GeeksForGeeks";

   // Static nested class
   public static class NestedStaticClass{

       // Only static members of Outer class is directly accessible in nested
       // static class
       public void printMessage() {

         // Try making 'message' a non-static variable, there will be
         // compiler error
         System.out.println("Message from nested static class: " + msg);
       }
   }

    // non-static nested class - also called Inner class
    public class InnerClass{

       // Both static and non-static members of Outer class are accessible in
       // this Inner class
       public void display(){
          System.out.println("Message from non-static nested class: "+ msg);
       }
    }
}
class Main
{
    // How to create instance of static and non static nested class?
    public static void main(String args[]){

       // create instance of nested Static class
       OuterClass.NestedStaticClass printer = new OuterClass.NestedStaticClass();

       // call non static method of nested static class
       printer.printMessage();

       // In order to create instance of Inner class we need an Outer class
       // instance. Let us create Outer class instance for creating
       // non-static nested class
       OuterClass outer = new OuterClass();
       OuterClass.InnerClass inner  = outer.new InnerClass();

       // calling non-static method of Inner class
       inner.display();

       // we can also combine above steps in one step to create instance of
       // Inner class
       OuterClass.InnerClass innerObject = new OuterClass().new InnerClass();

       // similarly we can now call Inner class method
       innerObject.display();
    }
}
```

Output:

```
Message from nested static class: GeeksForGeeks
Message from non-static nested class: GeeksForGeeks
Message from non-static nested class: GeeksForGeeks
```

Reference Book:
Introduction To Java Programming by Y. DANIEL LIANG

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Can we Overload or Override static methods in java ?

Let us first define Overloading and Overriding.

**Overriding** : Overriding is a feature of OOP languages like Java that is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class). The implementation to be executed is decided at run-time and decision is made according to the object used for call. Note that signatures of both methods must be same. Refer Overriding in Java for details.

**Overloading**: Overloading is also a feature of OOP languages like Java that is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters. Note that in both C++ and Java, methods cannot be overloaded according to return type.

**Can we overload static methods?**

The answer is 'Yes'. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program.

```
// filename Test.java
public class Test {
    public static void foo() {
        System.out.println("Test.foo() called ");
    }
    public static void foo(int a) {
        System.out.println("Test.foo(int) called ");
    }
    public static void main(String args[])
    {
        Test.foo();
        Test.foo(10);
    }
}
```

Output:

```
Test.foo() called
Test.foo(int) called
```

**Can we overload methods that differ only by static keyword?**

We cannot overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. This behaviour is same in C++ (See point 2 of this).

```
// filename Test.java
public class Test {
    public static void foo() {
        System.out.println("Test.foo() called ");
    }
    public void foo() { // Compiler Error: cannot redefine foo()
        System.out.println("Test.foo(int) called ");
    }
    public static void main(String args[]) {
        Test.foo();
    }
}
```

Output: Compiler Error, cannot redefine foo()

**Can we Override static methods in java?**

We can declare static methods with same signature in subclass, but it is not considered overriding as there won't be any run-time polymorphism. Hence the answer is 'No'.
If a derived class defines a static method with same signature as a static method in base class, the method in the derived class hides the method in the base class.

```
/* Java program to show that if static method is redefined by
   a derived class, then it is not overriding. */

// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // This method hides display() in Base
    public static void display() {
         System.out.println("Static or class method from Derived");
    }

    // This method overrides print() in Base
    public void print() {
         System.out.println("Non-static or Instance method from Derived");
   }
}

// Driver class
public class Test {
    public static void main(String args[ ])  {
       Base obj1 = new Derived();

       // As per overriding rules this should call to class Derive's static
       // overridden method. Since static method can not be overridden, it
       // calls Base's display()
       obj1.display();

       // Here overriding works and Derive's print() is called
       obj1.print();
    }
}
```

Output:

```
Static or class method from Base
Non-static or Instance method from Derived
```

Following are some important points for method overriding and static methods in Java.

**1)** For class (or static) methods, the method according to the type of reference is called, not according to the abject being referred, which means method call is decided at compile time.

**2)** For instance (or non-static) methods, the method is called according to the type of object being referred, not according to the type of reference, which means method calls is decided at run time.

**3)** An instance method cannot override a static method, and a static method cannot hide an instance method. For example, the following program has two compiler errors.

```
/* Java program to show that if static methods are redefined by
   a derived class, then it is not overriding but hidding. */
```

```
// Superclass
class Base {

    // Static method in base class which will be hidden in subclass
    public static void display() {
        System.out.println("Static or class method from Base");
    }

     // Non-static method which will be overridden in derived class
     public void print()  {
         System.out.println("Non-static or Instance method from Base");
    }
}

// Subclass
class Derived extends Base {

    // Static is removed here (Causes Compiler Error)
    public void display() {
        System.out.println("Non-static method from Derived");
    }

    // Static is added here (Causes Compiler Error)
    public static void print() {
        System.out.println("Static method from Derived");
   }
}
```

**4)** In a subclass (or Derived Class), we can overload the methods inherited from the superclass. Such overloaded methods neither hide nor override the superclass methods — they are new methods, unique to the subclass.

**References:**

http://docs.oracle.com/javase/tutorial/java/IandI/override.html

This article is contrubuted by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## “final” keyword in java

final keyword is used in Java in different contexts. The idea is make an entity non-modifiable. Following are different contexts where final is used.

**Final variables** A final variable can only be assigned once. If the variable is a reference, this means that the variable cannot be re-bound to reference another object.

```
// The following doesn't compile because final variable is assigned
// value twice.
class Main {
   public static void main(String args[]){
      final int i = 20;
      i = 30;
   }
}
```

Output

```
Compiler Error: cannot assign a value to final variable i
```

Note the difference between C++ const variables and Java final variables. const variables in C++ must be assigned a value when declared. For final variables n Java, it is not necessary. A final variable can be assigned value later, but only once. For example see the following Java program.

```
// The following program compiles and runs fine
class Main {
   public static void main(String args[]){
      final int i;
      i = 20;
      System.out.println(i);
   }
}
```

Output:

```
20
```

**Final classes** A final class cannot be extended (inherited)

```
//Error in following program as we are trying to extend a final class
final class Base { }
class Derived extends Base { }

public class Main {
   public static void main(String args[]) {
   }
}
```

Output:

```
 Compiler Error: cannot inherit from final Base
```

**Final methods** A final method cannot be overridden by subclasses.
//Error in following program as we are trying to override a final method.

```
class Base {
  public final void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {  // Error
       System.out.println("Derived::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

Output:

```
Compiler Error: show() in Derived cannot override show()
in Base overridden method is final
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# Super Keyword in Java

The **super** keyword in java is a reference variable that is used to refer parent class objects.  The keyword “super” came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

**1. Use of super with variables:** This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */
class Vehicle
{
    int maxSpeed = 120;
}

/* sub class Car extending vehicle */
class Car extends Vehicle
{
    int maxSpeed = 180;

    void display()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: " + super.maxSpeed);
    }
}

/* Driver program to test */
class Test
{
    public static void main(String[] args)
    {
        Car small = new Car();
        small.display();
    }
}
```

Output:

```
Maximum Speed: 120
```

In the above example, both base class and subclass have a member maxSpeed. We could access maxSpeed of base class in sublcass using super keyword.

**2. Use of super with methods:** This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
class Person
{
    void message()
    {
        System.out.println("This is person class");
    }
}

/* Subclass Student */
class Student extends Person
{
    void message()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    void display()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
class Test
{
    public static void main(String args[])
    {
        Student s = new Student();

        // calling display() of Student
        s.display();
    }
}
```

Output:

```
This is student class
This is person class
```

In the above example, we have seen that if we only call method message() then, the current class message() is invoked but with the use of super keyword, message() of superclass could also be invoked.

3. **Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, ''super' can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
class Person
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
class Student extends Person
{
    Student()
    {
        // invoke or call parent class constructor
        super();

        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
class Test
{
    public static void main(String[] args)
    {
        Student s = new Student();
    }
}
```

Output:

```
Person class Constructor
Student class Constructor
```

In the above example we have called the superclass constructor using keyword 'super' via subclass constructor.

**Other Important points:**

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error. Object *does* have such a constructor, so if Object is the only superclass, there is no problem.
3. If a subclass constructor invokes a constructor of its superclass, either explicitly or implicitly, you might think that a whole chain of constructors called, all the way back to the constructor of Object. This, in fact, is the case. It is called *constructor chaining.*.

This article is contributed by **Vishwajeet Srivastava**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

# Blank Final in Java

A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later.

```
    final int i = 10;
    i = 30; // Error because i is final.
```

A **blank final** variable in Java is a final variable that is not initialized during declaration. Below is a simple example of blank final.

```
    // A simple blank final example
    final int i;
    i = 30;
```

**How are values assigned to blank final members of objects?**
Values must be assigned in constructor.

```
// A sample Java program to demonstrate use and
// working of blank final
class Test
{
    // We can initialize here, but if we
    // initialize here, then all objects get
    // the same value.  So we use blank final
    final int i;

    Test(int x)
    {
        // Since we have initialized above, we
        // must initialize i in constructor.
        // If we remove this line, we get compiler
        // error.
        i = x;
    }
}
```

```
// Driver Code
class Main
{
    public static void main(String args[])
    {
        Test t1 = new Test(10);
        System.out.println(t1.i);

        Test t2 = new Test(20);
        System.out.println(t2.i);
    }
}
```

**Output:**

```
10
20
```

***If we have more than one constructors or overloaded constructor in class, then blank final variable must be initialized in all of them. However constructor chaining can be used to initialize the blank final variable.***

```
// A Java program to demonstrate that we can
// use constructor chaining to initialize
// final members
class Test
{
    final public int i;

    Test(int val)    {  this.i = val;  }

    Test()
    {
        // Calling Test(int val)
        this(10);
    }

    public static void main(String[] args)
    {
        Test t1 = new Test();
        System.out.println(t1.i);

        Test t2 = new Test(20);
        System.out.println(t2.i);
    }
}
```

**Output:**

```
10
20
```

Blank final variables are used to create immutable objects (objects whose members can't be changed once initialized).

This article is contributed by **Himanshi Gupta.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Java

## final, finally and finalize in Java

**final:** The final keyword in java has different meaning depending upon it is applied to variable, class or method.

- *Variables:* The value cannot be changed once initialized.
- *Method:* The method cannot be overridden by a subclass.
- *Class:* The class cannot be subclassed.

See final keyword in Java for Code examples of above three.

**finally:** The finally keyword is used in association with a try/catch block and guarantees that a section of code will be executed, even if an exception is thrown. The finally block will be executed after the try and catch blocks, but before control transfers back to its origin.

```
// A Java program to demonstrate finally.
class Geek
{
    // A method that throws an exception and has finally.
    // This method will be called inside try-catch.
    static void  A()
    {
        try
        {
            System.out.println("inside A");
            throw new RuntimeException("demo");
        }
        finally
        {
            System.out.println("A's finally");
        }
    }

    // This method also calls finally. This method
    // will be called outside try-catch.
    static void  B()
    {
        try
        {
            System.out.println("inside B");
            return;
        }
        finally
        {
            System.out.println("B's finally");
        }
    }

    public static void main(String args[])
    {
        try
        {
            A();
        }
        catch (Exception e)
        {
            System.out.println("Exception caught");
        }
        B();
    }
}
```

**Output:**

```
inside A
A's finally
Exception caught
inside B
B's finally
```

**finalize:** The automatic garbage collector call the finalize() method just before actually destroying the object. A class can therefore override the finalize() method from the Object class in order to define custom behavior during garbage collection.Syntax:

```
protected void finalize() throws Throwable
{
    /* close open files, release resources, etc */
}
```

Note that there is no guarantee about the time when finalize is called. It may be called any time after the object is not being referred anywhere (cab be garbage collected).

This article is contributed by **Manav**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# How are parameters passed in Java?

**See this for detailed description.**

In Java, parameters are always passed by value. For example, following program prints i = 10, j = 20.

```
//  Test.java
class Test {
   // swap() doesn't swap i and j
   public static void swap(Integer i, Integer j) {
      Integer temp = new Integer(i);
      i = j;
      j = temp;
   }
   public static void main(String[] args) {
      Integer i = new Integer(10);
      Integer j = new Integer(20);
      swap(i, j);
      System.out.println("i = " + i + ", j = " + j);
   }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Returning Multiple values in Java

Java doesn't support multi-value returns. We can use following solutions to return multiple values.

**If all returned elements are of same type**

We can return an array in Java. Below is a Java program to demonstrate the same.

```
// A Java program to demonstrate that a method
// can return multiple values of same type by
// returning an array
class Test
{
    // Returns an array such that first element
    // of array is a+b, and second element is a-b
    static int[] getSumAndSub(int a, int b)
    {
        int[] ans = new int[2];
        ans[0] = a + b;
        ans[1] = a - b;

        // returning array of elements
        return ans;
    }

    // Driver method
    public static void main(String[] args)
    {
        int[] ans = getSumAndSub(100,50);
        System.out.println("Sum = " + ans[0]);
        System.out.println("Sub = " + ans[1]);
    }
}
```

The output of the above code will be:

```
Sum = 150
Sub = 50
```

**If returned elements are of different types**

We can be encapsulate all returned types into a class and then return an object of that class.
Let us have a look at the following code.

```
// A Java program to demonstrate that we can return
// multiple values of different types by making a class
// and returning an object of class.

// A class that is used to store and return
// two members of different types
class MultiDiv
{
    int mul;    // To store multiplication
    double div; // To store division
    MultiDiv(int m, double d)
    {
        mul = m;
        div = d;
    }
}

class Test
{
    static MultiDiv getMultandDiv(int a, int b)
    {
        // Returning multiple values of different
        // types by returning an object
        return new MultiDiv(a*b, (double)a/b);
    }

    // Driver code
    public static void main(String[] args)
    {
        MultiDiv ans = getMultandDiv(10, 20);
        System.out.println("Multiplication = " + ans.mul);
        System.out.println("Division = " + ans.div);
    }
}
```

Output:

```
 Multiplication = 200
Division = 0.5
```

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Overloading in Java

Overloading allows different methods to have same name, but different signatures where signature can differ by number of input parameters or type of input parameters or both. Overloading is related to compile time (or static) polymorphism.

```
// Java program to demonstrate working of method
// overloading in Java.

public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y) {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z) {
         return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y) {
         return (x + y);
    }

    // Driver code
    public static void main(String args[]) {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

```
30
60
31.0
```

**What is the advantage?**
We don't have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2,

… or sum2Int, sum3Int, … etc.

**Can we overload methods on return type?**

We **cannot** overload by return type. This behavior is same in C++. Refer this for details

```
public class Main
{
    public int foo() { return 10; }

    // compiler error: foo() is already defined
    public char foo() { return 'a'; }

    public static void main(String args[])
    {
    }
}
```

**Can we overload static methods?**

The answer is '**Yes**'. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program. Refer this for details.

**Can we overload methods that differ only by static keyword?**

We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. Refer this for details.

**Can we overload main() in Java?**

Like other static methods, we **can** overload main() in Java. Refer overloading main() in Java for more details.

```
// A Java program with overloaded main()
import java.io.*;

public class Test {

    // Normal main()
    public static void main(String[] args) {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }

    // Overloaded main methods
    public static void main(String arg1) {
        System.out.println("Hi, " + arg1);
        Test.main("Dear Geek","My Geek");
    }
    public static void main(String arg1, String arg2) {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
}
```

Output :

```
Hi Geek (from main)
Hi, Geek
Hi, Dear Geek, My Geek
```

**Does Java support Operator Overloading?**

Unlike C++, Java doesn't allow user defined overloaded operators. Internally Java overloads operators, for example + is overloaded for concatenation.

**What is the difference between Overloading and Overriding?**

- Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.
  OverridingVsOverloading
- Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.

**Related Articles:**

- Different ways of Method Overloading in Java
- Method Overloading and Null error in Java
- Can we Overload or Override static methods in java ?

This article is contributed by **Shubham Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Function overloading and return type

In C++ and Java, functions can not be overloaded if they differ only in the return type.

For example, the following program C++ and Java programs fail in compilation.

**C++ Program**

```
#include<iostream>
int foo() {
    return 10;
}

char foo() {  // compiler error; new declaration of foo()
    return 'a';
}

int main()
{
    char x = foo();
    getchar();
    return 0;
}
```

**Java Program**

```
// filename Main.java
public class Main {
    public int foo() {
        return 10;
    }
    public char foo() { // compiler error: foo() is already defined
        return 'a';
    }
    public static void main(String args[])
    {
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

C/C++ Puzzles
Java
Java

# Overriding equals method in Java

Consider the following Java program:

```
class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(10, 15);
        if (c1 == c2) {
            System.out.println("Equal ");
        } else {
            System.out.println("Not Equal ");
        }
    }
}
```

Output:

```
Not Equal
```

The reason for printing “Not Equal” is simple: when we compare c1 and c2, it is checked whether both c1 and c2 refer to same object or not (object variables are always references in Java). c1 and c2 refer to two different objects, hence the value (c1 == c2) is false. If we create another reference say c3 like following, then (c1 == c3) will give true.

```
Complex c3 = c1;  // (c3 == c1) will be true
```

So, how do we check for equality of values inside the objects? All classes in Java inherit from the Object class, directly or indirectly (See point 1 of this). The Object class has some basic methods like clone(), toString(), equals(),.. etc. We can override the equals method in our class to check whether two objects have same data or not.

```
class Complex {

    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // Overriding equals() to compare two Complex objects
    @Override
    public boolean equals(Object o) {

        // If the object is compared with itself then return true
        if (o == this) {
            return true;
        }

        /* Check if o is an instance of Complex or not
          "null instanceof [type]" also returns false */
        if (!(o instanceof Complex)) {
            return false;
        }

        // typecast o to Complex so that we can compare data members
        Complex c = (Complex) o;

        // Compare the data members and return accordingly
        return Double.compare(re, c.re) == 0
                && Double.compare(im, c.im) == 0;
    }
}

// Driver class to test the Complex class
```

```
public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(10, 15);
        if (c1.equals(c2)) {
            System.out.println("Equal ");
        } else {
            System.out.println("Not Equal ");
        }
    }
}
```

Output:

```
Equal
```

As a side note, when we override equals(), it is recommended to also override the hashCode() method. If we don't do so, equal objects may get different hash-values; and hash based collections, including HashMap, HashSet, and Hashtable do not work properly (see this for more details). We will be coverig more about hashCode() in a separate post.

References:
Effective Java Second Edition

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Overriding toString() in Java

This post is similar to Overriding equals method in Java. Consider the following Java program:

```
// file name: Main.java

class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

Output:

```
Complex@19821f
```

The output is, *class name, then 'at' sign, and at the end hashCode of object.* All classes in Java inherit from the Object class, directly or indirectly (See point 1 of this). The Object class has some basic methods like clone(), toString(), equals(),.. etc. The default toString() method in Object prints "class name @ hash code". We can override toString() method in our class to print proper output. For example, in the following code toString() is overridden to print "Real + i Imag" form.

```
// file name: Main.java

class Complex {
    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    /* Returns the string representation of this Complex number.
       The format of string is "Re + iIm" where Re is real part
       and Im is imagenary part.*/
    @Override
    public String toString() {
        return String.format(re + " + i" + im);
    }
}

// Driver class to test the Complex class
public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        System.out.println(c1);
    }
}
```

Output:

```
10.0 + i15.0
```

In general, it is a good idea to override toString() as we get get proper output when an object is used in print() or println().

**References:**
Effective Java by Joshua Bloch

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

## Private and final methods in Java

When we use *final* specifier with a method, the method cannot be overridden in any of the inheriting classes. Methods are made final due to design reasons.

Since private methods are inaccessible, they are implicitly final in Java. So adding *final* specifier to a private method doesn't add any value. It may in-fact cause unnecessary confusion.

```
class Base {

   private final void foo() {}

   // The above method foo() is same as following. The keyword
   // final is redundant in above declaration.

   // private void foo() {}
}
```

For example, both 'program 1' and 'program 2' below produce same compiler error "foo() has private access in Base".

**Program 1**

```
// file name: Main.java
class Base {
    private final void foo() {}
}

class Derived extends Base {
    public void foo() {}
}

public class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.foo();
    }
}
```

**Program 2**

```
// file name: Main.java
class Base {
    private void foo() {}
}

class Derived extends Base {
    public void foo() {}
}

public class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.foo();
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Java is Strictly Pass by Value!

Consider the following Java program that passes a **primitive type** to function.

```
public class Main
{
    public static void main(String[] args)
    {
        int x = 5;
        change(x);
        System.out.println(x);
    }
    public static void change(int x)
    {
        x = 10;
    }
}
```

Output:

```
5
```

We pass an int to the function "change()" and as a result the change in the value of that integer is not reflected in the main method. Like C/C++, Java creates a copy of the variable being passed in the method and then do the manipulations. Hence the change is not reflected in the main method.

**How about objects or references?**

In Java, all primitives like int, char, etc are similar to C/C++, but all non-primitives (or objects of any class) are always references. So it gets tricky when we pass object references to methods. Java creates a copy of references and pass it to method, but they still point to same memory reference. Mean if set some other object to reference passed inside method, the object from calling method as well its reference will remain unaffected.

**The changes are not reflected back if we change the object itself to refer some other location or object**

If we assign reference to some other location, then changes are not reflected back in main().

```
// A Java program to show that references are also passed
// by value.
class Test
{
    int x;
    Test(int i) { x = i; }
    Test()      { x = 0; }
}

class Main
{
    public static void main(String[] args)
    {
        // t is a reference
        Test t = new Test(5);

        // Reference is passed and a copy of reference
        // is created in change()
        change(t);

        // Old value of t.x is printed
        System.out.println(t.x);
    }
    public static void change(Test t)
    {
        // We changed reference to refer some other location
        // now any changes made to reference are not reflected
        // back in main
        t = new Test();

        t.x = 10;
    }
}
```

Output:

```
5
```

**Changes are reflected back if we do not assign reference to a new location or object:**
If we do not change the reference to refer some other object (or memory location), we can make changes to the members and these changes are reflected back.

```
// A Java program to show that we can change members using using
// reference if we do not change the reference itself.
class Test
{
    int x;
    Test(int i) { x = i; }
    Test()      { x = 0; }
}

class Main
{
    public static void main(String[] args)
    {
        // t is a reference
        Test t = new Test(5);

        // Reference is passed and a copy of reference
        // is created in change()
        change(t);

        // New value of x is printed
        System.out.println(t.x);
    }

    // This change() doesn't change the reference, it only
    // changes member of object referred by reference
    public static void change(Test t)
    {
        t.x = 10;
    }
}
```

Output:

```
10
```

**Exercise:** Predict the output of following Java program

```
//  Test.java
class Main {
   // swap() doesn't swap i and j
   public static void swap(Integer i, Integer j)
   {
      Integer temp = new Integer(i);
      i = j;
      j = temp;
   }
   public static void main(String[] args)
   {
      Integer i = new Integer(10);
      Integer j = new Integer(20);
      swap(i, j);
      System.out.println("i = " + i + ", j = " + j);
   }
}
```

This article is contributed by **Pranjal Mathur**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## GFact 48 | Overloading main() in Java

Consider the below Java program.

```
// A Java program with overloaded main()
import java.io.*;

public class Test {

    // Normal main()
    public static void main(String[] args) {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }
    // Overloaded main methods
    public static void main(String arg1) {
        System.out.println("Hi, " + arg1);
        Test.main("Dear Geek","My Geek");
    }
    public static void main(String arg1, String arg2) {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
}
```

**Output:**

```
Hi Geek (from main)
Hi, Geek
Hi, Dear Geek, My Geek
```

**Important Points:**
The main method in Java is no extra-terrestrial method. Apart from the fact that main() is just like any other method & can be overloaded in a similar manner, JVM always looks for the method signature to launch the program.

- The normal main method acts as an entry point for the JVM to start the execution of program.
- We can overload the main method in Java. But the program doesn't execute the overloaded main method when we run your program, we need to call the overloaded main method from the actual main method only.

**Related Articles :**
Valid variants of main() in Java
Overload main in C++
Can we Overload or Override static methods in java ?

This article is contributed by **Himanshi Gupta.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Post navigation
Next Post >>

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Load Comments
Share this post!

# Valid variants of main() in Java

Below are different variants of main() that are valid.

1. **Default prototype:** Below is the most common way to write main() in Java.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

   Output:

```
Main Method
```

   Meaning of the main Syntax:

```
public:  JVM can execute the method from anywhere.
static:  Main method can be called without object.
void:    The main method doesn't return anything.
main():  Name configured in the JVM.
String[]: Accepts the command line arguments.
```

2. **Order of Modifiers:** We can swap positions of static and public in main().

```
class Test
{
    static public void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

   Output:

```
Main Method
```

3. **Variants of String array arguments:** We can place square brackets at different positions and we can use varargs (…) for string parameter.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

   Output:

```
Main Method
```

```
class Test
{
    public static void main(String []args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

```
class Test
{
    public static void main(String args[])
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

```
class Test
{
    public static void main(String...args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

4. **Final Modifier String argument:** We can make String args[] as final.

```
class Test
{
    public static void main(final String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

5. **Final Modifier to static main method:** We can make main() as final.

```
class Test
{
    public final static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

6. **synchronized keyword to static main method:** We can make main() synchronized.

```
class Test
{
    public synchronized static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

7. **strictfp keyword to static main method:** strictfp can be used to restrict floating point calculations.

```
class Test
{
    public strictfp static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

8. **Combinations of all above keyword to static main method:**

```
class Test
{
    final static synchronized strictfp static void main(String[] args)
    {
        System.out.println("Main Method");
    }
}
```

Output:

```
Main Method
```

9. **Overloading Main method:** We can overload main() with different types of parameters.

```
class Test
{
    public static void main(String[] args)
    {
        System.out.println("Main Method String Array");
    }
    public static void main(int[] args)
    {
        System.out.println("Main Method int Array");
    }
}
```

Output:

```
Main Method String Array
```

10. **Inheritance of Main method:** JVM Executes the main() without any errors.

```
class A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Parent");
    }
}

class B extends A
{

}
```

Two class files, A.class and B.class are generated by compiler. When we execute any of the two .class, JVM executes with no error.

```
O/P: Java A
Main Method Parent
O/P: Java B
Main Method Parent
```

11. **Method Hiding of main(), but not Overriding:** Since main() is static, derived class main() hides the base class main. (See Shadowing of static functions for details.)

```
class A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Parent");
    }
}
class B extends A
{
    public static void main(String[] args)
    {
        System.out.println("Main Method Child");
    }
}
```

Two classes, A.class and B.class are generated by Java Compiler javac. When we execute both the .class, JVM executes with no error.

```
O/P: Java A
Main Method Parent
O/P: Java B
Main Method Child
```

This article is contributed by **Mahesh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

# Variable Arguments (Varargs) in Java

Variable arguments (or varargs) were introduced in JDK 5. Below is a code snippet for illustrating the concept.

```
// A simple Java program to demonstrate varargs
class Test1
{
    // A method that takes variable number of intger
    // arguments.
    static void fun(int ...a)
    {
        System.out.println("Number of arguments is: " + a.length);

        // using for each loop to display contents of a
        for (int i: a)
            System.out.print(i + " ");
        System.out.println();
    }

    // Driver code
    public static void main(String args[])
    {
        // Calling the varargs method with different number
        // of parameters
        fun(100);         // one parameter
        fun(1, 2, 3, 4);  // four parameters
        fun();            // no parameter
    }
}
```

Output:

```
Number of arguments is: 1
100
Number of arguments is: 4
1 2 3 4
Number of arguments is: 0
```

**Explanation of above program :**

- The … syntax tells the compiler that varargs has been used and these arguments should be stored in the **array referred to by a**.

- The variable **a** is operated on as an array. In this case, we have defined the data type of a as int. So it can take only integer values. The number of arguments can be found out using a.length, the way we find the length of an array in Java.

**A method can have normal parameters along with the variable length parameters** but one should ensure that there **exists only one varargs parameter** that should be written last in the parameter list of the method declaration.

```
int nums(int a, float b, double … c)
```

In this case, the first two arguments are matched with the first two parameters and the remaining arguments belong to c.

An example program showcasing the use of varargs along with normal parameters :

```
// A simple Java program to demonstrate varargs with normal
// arguments
class Test2
{
    // Takes string as a argument followed by varargs
    static void fun2(String str, int ...a)
    {
        System.out.println("String: " + str);
        System.out.println("Number of arguments is: "+ a.length);

        // using for each loop to display contents of a
        for (int i: a)
            System.out.print(i + " ");

        System.out.println();
    }

    public static void main(String args[])
    {
        // Calling fun2() with different parameter
        fun2("GeeksforGeeks", 100, 200);
        fun2("CSPortal", 1, 2, 3, 4, 5);
        fun2("forGeeks");
    }
}
```

```
String: GeeksforGeeks
Number of arguments is: 2
100 200
String: CSportal
Number of arguments is: 5
1 2 3 4 5
String: forGeeks
Number of arguments is: 0
```

Below are some important facts about varargs.

- Vararg Methods can also be overloaded but overloading may lead to ambiguity.
- Prior to JDK 5, variable length arguments could be handled into two ways : One was using overloading, other was using array argument.
- There can be only one variable argument in a method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that are **erroneous:**

```
void method(String... gfg, int... q)
{
    // Compile time error as there are two
    // varargs
}
```

```
void method(int... gfg, String q)
{
    // Compile time error as vararg appear
    // before normal argument
}
```

This article is contributed by **Niraj Srimal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Primitive Wrapper Classes are Immutable in Java

Consider below Java program.

```
// Java program to demonstrate that prmitive
// wrapper classes are immutable
class Demo
{
    public static void main(String[] args)
    {
        Integer i = new Integer(12);
        System.out.println(i);
        modify(i);
        System.out.println(i);
    }

    private static void modify(Integer i)
    {
        i = i + 1;
    }
}
```

Output :

```
12
12
```

The parameter i is reference in modify and refers to same object as i in main(), but changes made to i are not reflected in main(), why?

**Explanation:**

All primitive wrapper classes (Integer, Byte, Long, Float, Double, Character, Boolean and Short) are immutable in Java, so operations like addition and subtraction create a new object and not modify the old.

The below line of code in the modify method is operating on wrapper class Integer, not an int

**i = i + 1;**

It does the following:

1. Unbox i to an int value
2. Add 1 to that value
3. Box the result into another Integer object
4. Assign the resulting Integer to i (thus changing what object i references)

Since object references are passed by value, the action taken in the modify method does not change i that was used as an argument in the call to modify. Thus the main routine still prints 12 after the method returns.

This article is contributed by **Yogesh D Doshi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

# Clone() method in Java

Object cloning refers to creation of exact copy of an object. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

**Using Assignment Operator to create copy of reference variable**

In Java, there is no operator to create copy of an object. Unlike C++, in Java, if we use assignment operator then it will create a copy of reference variable and not the object. This can be explained by taking an example. Following program demonstrates the same.

```
// Java program to demonstrate that assignment
// operator only creates a new reference to same
// object.
import java.io.*;

// A test class whose objects are cloned
class Test
{
    int x, y;
    Test()
    {
        x = 10;
        y = 20;
    }
}

// Driver Class
class Main
{
    public static void main(String[] args)
    {
         Test ob1 = new Test();

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

         // Creating a new reference variable ob2
         // pointing to same address as ob1
         Test ob2 = ob1;

         // Any change made in ob2 will be reflected
         // in ob1
         ob2.x = 100;

         System.out.println(ob1.x+" "+ob1.y);
         System.out.println(ob2.x+" "+ob2.y);
    }
}
```

**Output:**

```
10 20
100 20
100 20
```

**Creating a copy using clone() method**

The class whose object's copy is to be made must have a public clone method in it or in one of its parent class.

- Every class that implements clone() should call super.clone() to obtain the cloned object reference.
- The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class's object.
- Syntax:

  ```
  protected Object clone() throws CloneNotSupportedException
  ```

**Usage of clone() method -Shallow Copy**

```
// A Java program to demonstrate shallow copy
// using clone()
import java.util.ArrayList;

// An object reference of this class is
// contained by Test2
class Test
{
    int x, y;
}

// Contains a reference of Test and implements
// clone with shallow copy.
class Test2 implements Cloneable
{
    int a;
    int b;
    Test c = new Test();
    public Object clone() throws
                CloneNotSupportedException
```

```
    {
        return super.clone();
    }
}

// Driver class
public class Main
{
    public static void main(String args[]) throws
                          CloneNotSupportedException
    {
       Test2 t1 = new Test2();
       t1.a = 10;
       t1.b = 20;
       t1.c.x = 30;
       t1.c.y = 40;

       Test2 t2 = (Test2)t1.clone();

       // Creating a copy of object t1 and passing
       //  it to t2
       t2.a = 100;

       // Change in primitive type of t2 will not
       // be reflected in t1 field
       t2.c.x = 300;

       // Change in object type field will be
       // reflected in both t2 and t1(shallow copy)
       System.out.println(t1.a + " " + t1.b + " " +
                          t1.c.x + " " + t1.c.y);
       System.out.println(t2.a + " " + t2.b + " " +
                          t2.c.x + " " + t2.c.y);
    }
}
```

**Output:**

```
10 20 300 40
100 20 300 40
```

In the above example, t1.clone returns the shallow copy of the object t1. To obtain a deep copy of the object certain modifications have to be made in clone method after obtaining the copy.

**Deep Copy vs Shallow Copy**

- **Shallow copy** is method of copying an object and is followed by default in cloning. In this method the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
- Therefore, any changes made in referenced objects in object X or Y will be reflected in other object.

*Shallow copies are cheap and simple to make. In above example, we created a shallow copy of object.*

**Usage of clone() method – Deep Copy**

- If we want to create a deep copy of object X and place it in a new object Y then new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In below example, we create a deep copy of object.
- A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

```
// A Java program to demonstrate deep copy
// using clone()
import java.util.ArrayList;

// An object reference of this class is
// contained by Test2
class Test
{
    int x, y;
}


// Contains a reference of Test and implements
// clone with deep copy.
class Test2 implements Cloneable
{
    int a, b;

    Test c = new Test();

    public Object clone() throws
                CloneNotSupportedException
    {
        // Assign the shallow copy to new refernce variable t
        Test2 t = (Test2)super.clone();

        t.c = new Test();

        // Create a new object for the field c
        // and assign it to shallow copy obtained,
        // to make it a deep copy
        return t;
    }
}

public class Main
{
    public static void main(String args[]) throws
                           CloneNotSupportedException
    {
       Test2 t1 = new Test2();
       t1.a = 10;
       t1.b = 20;
       t1.c.x = 30;
       t1.c.y = 40;

       Test2 t3 = (Test2)t1.clone();
       t3.a = 100;

       // Change in primitive type of t2 will not
       // be reflected in t1 field
       t3.c.x = 300;

       // Change in object type field of t2 will not
```

```
        // be reflected in t1(deep copy)
        System.out.println(t1.a + " " + t1.b + " " +
                     t1.c.x + " " + t1.c.y);
        System.out.println(t3.a + " " + t3.b + " " +
                     t3.c.x + " " + t3.c.y);
    }
}
```

**Output:**

```
10 20 30 40
100 20 300 0
```

In the above example, we can see that a new object for Test class has been assigned to copy object that will be returned in clone method.Due to this t3 will obtain a deep copy of the object t1. So any changes made in 'c' object fields by t3 ,will not be reflected in t1.

**Advantages of clone method:**

- If we use assignment operator to assign an object reference to another reference variable then it will point to same address location of the old object and no new copy of the object will be created. Due to this any changes in reference variable will be reflected in original object.
- If we use copy constructor, then we have to copy all of the data over explicitly i.e. we have to reassign all the fields of the class in constructor explicitly. But in clone method this work of creating a new copy is done by the method itself.So to avoid extra processing we use object cloning.

This article is contributed by **Ankit Agarwal.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# Remote Method Invocation in Java

Remote Method Invocation (RMI) is an API which allows an object to invoke a method on an object that exists in another address space, which could be on the same machine or on a remote machine. Through RMI, object running in a JVM present on a computer (Client side) can invoke methods on an object present in another JVM (Server side). RMI creates a public remote server object that enables client and server side communications through simple method calls on the server object.

**Working of RMI**

The communication between client and server is handled by using two intermediate objects: Stub object (on client side) and Skeleton object (on server side).

**Stub Object**

The stub object on the client machine builds an information block and sends this information to the server. The block consists of

- An identifier of the remote object to be used
- Method name which is to be invoked
- Parameters to the remote JVM

**Skeleton Object**

The skeleton object passes the request from the stub object to the remote object. It performs following tasks

- It calls the desired method on the real object present on the server.

RMI in Java

- It forwards the parameters received from the stub object to the method.

**Steps to implement Interface**

1. Defining a remote interface
2. Implementing the remote interface
3. Creating Stub and Skeleton objects from the implementation class using rmic (rmi complier)
4. Start the rmiregistry
5. Create and execute the server application program
6. Create and execute the client application program.

**Step 1: Defining the remote interface**

The first thing to do is to create an interface which will provide the description of the methods that can be invoked by remote clients. This interface should extend the Remote interface and the method prototype within the interface should throw the RemoteException.

```
// Creating a Search interface
import java.rmi.*;
public interface Search extends Remote
{
    // Declaring the method prototype
    public String query(String search) throws RemoteException;
}
```

**Step 2: Implementing the remote interface**

The next step is to implement the remote interface. To implement the remote interface, the class should extend to UnicastRemoteObject class of java.rmi package. Also, a default constructor needs to be created to throw the java.rmi.RemoteException from its parent constructor in class.

```
// Java program to implement the Search interface
import java.rmi.*;
import java.rmi.server.*;
public class SearchQuery extends UnicastRemoteObject
                         implements Search
{
    // Default constructor to throw RemoteException
    // from its parent constructor
    SearchQuery() throws RemoteException
    {
        super();
    }

    // Implementation of the query interface
    public String query(String search)
                       throws RemoteException
    {
        String result;
        if (search.equals("Reflection in Java"))
            result = "Found";
        else
            result = "Not Found";

        return result;
    }
}
```

**Step 3: Creating Stub and Skeleton objects from the implementation class using rmic**

The rmic tool is used to invoke the rmi compiler that creates the Stub and Skeleton objects. Its prototype is rmic classname. For above program the following command need to be executed at the command prompt
rmic SearchQuery

**STEP 4: Start the rmiregistry**

Start the registry service by issuing the following command at the command prompt start rmiregistry

**STEP 5: Create and execute the server application program**

The next step is to create the server application program and execute it on a separate command prompt.

- The server program uses createRegistry method of LocateRegistry class to create rmiregistry within the server JVM with the port number passed as argument.
- The rebind method of Naming class is used to bind the remote object to the new name.

```
//program for server application
import java.rmi.*;
import java.rmi.registry.*;
public class SearchServer
{
    public static void main(String args[])
    {
        try
        {
            // Create an object of the interface
            // implementation class
            Search obj = new SearchQuery();

            // rmiregistry within the server JVM with
            // port number 1900
            LocateRegistry.createRegistry(1900);

            // Binds the remote object by the name
            // geeksforgeeks
            Naming.rebind("rmi://localhost:1900"+
                          "/geeksforgeeks",obj);
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

**Step 6: Create and execute the client application program**

The last step is to create the client application program and execute it on a separate command prompt .The lookup method of Naming class is used to get the reference of the Stub object.

```
//program for client application
import java.rmi.*;
public class ClientRequest
{
    public static void main(String args[])
    {
        String answer,value="Reflection in Java";
        try
        {
            // lookup method to find reference of remote object
            Search access =
                (Search)Naming.lookup("rmi://localhost:1900"+
                                      "/geeksforgeeks");
            answer = access.query(value);
            System.out.println("Article on " + value +
                            " " + answer+" at GeeksforGeeks");
        }
        catch(Exception ae)
        {
            System.out.println(ae);
        }
    }
}
```

*Note: The above client and server program is executed on the same machine so localhost is used. In order to access the remote object from another machine, localhost is to be replaced with the IP address where the remote object is present.*

**Important Observations:**

1. RMI is a pure java solution to Remote Procedure Calls (RPC) and is used to create distributed application in java.
2. Stub and Skeleton objects are used for communication between client and server side.

This article is contributed by **Aakash Ojha**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# Default Methods In Java

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.

```
// A simple program to Test Interface default
// methods in java
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method Executed");
    }
}

class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // default method executed
        d.show();
    }
}
```

Output:

```
16
Default Method Executed
```

The default methods were introduced to provide backward comparability so that existing intefaces can use the lambda expressions without implementing the methods in the implementation class. Default methods are also known as **defender methods** or **virtual extension methods**.

**Static Methods:**
The interfaces can have static methods as well which is similar to static method of classes.

```
// A simple Java program to TestClassnstrate static
// methods in java
interface TestInterface
{
    // abstract method
    public void square (int a);

    // static method
    static void show()
    {
        System.out.println("Static Method Executed");
    }
}

class TestClass implements TestInterface
{
    // Implementation of square abstract method
    public void square (int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // Static method executed
        TestInterface.show();
    }
}
```

Output:

```
16
Static Method Executed
```

**Default Methods and Multiple Inheritance**
In case both the implemented interfaces contain deafult methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

```
 // A simple Java program to demonstrate multiple
// inheritance through default methods.
interface TestInterface1
{
    // default method
    default void show()
    {
        System.out.println("Default TestInterface1");
    }
}

interface TestInterface2
{
    // Default method
    default void show()
    {
        System.out.println("Default TestInterface2");
    }
}

// Implementation class code
class TestClass implements TestInterface1, TestInterface2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of TestInterface1 interface
        TestInterface1.super.show();

        // use super keyword to call the show
        // method of TestInterface2 interface
        TestInterface2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

```
Default TestInterface1
Default TestInterface2
```

**Important Points:**

1. Interfaces can have default methods with implementation from java 8 onwards.
2. Interfaces can have static methods as well similar to static method of classes.
3. Default methods were introduced to provide backward comparability for old interfaces so that they can have new methods without effecting existing code.

This article is contributed by **Akash Ojha**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

## Method overloading and null error in Java

In Java it is very common to overload methods in Java. Below is an interesting Java program.

```
public class Test
{
    // Overloaded methods
    public void fun(Integer i)
    {
        System.out.println("fun(Integer ) ");
    }
    public void fun(String name)
    {
        System.out.println("fun(String ) ");
    }

    // Driver code
    public static void main(String [] args)
    {
        Test mv = new Test();

        // This line causes error
        mv.fun(null);
    }
}
```

Output :

```
22: error: reference to fun is ambiguous
        mv.fun(null);
          ^
  both method fun(Integer) in Test and method fun(String) in Test match
1 error
```

The reason why we get compile time error in the above scenario is, here the method arguments Integer and String both are not primitive data types in Java. That means they accept null values. When we pass a null value to the method1 the compiler gets confused which method it has to select, as both are accepting the null.
This compile time error wouldn't happen unless we intentionally pass null value. For example see the below scenario which we follow generally while coding.

```
public class Test
{
    // Overloaded methods
    public void fun(Integer i)
    {
        System.out.println("fun(Integer ) ");
    }
    public void fun(String name)
    {
        System.out.println("fun(String ) ");
    }

    // Driver code
    public static void main(String [] args)
    {
        Test mv = new Test();

        Integer arg = null;

        // No compiler error
        mv.fun(arg);
    }
}
```

Output :

```
fun(Integer )
```

In the above scenario if the “arg” value is null due to the result of the expression, then the null value is passed to method1. Here we wouldn’t get compile time error because we are specifying that the argument is of type Integer, hence the compiler selects the method1(Integer i) and will execute the code inside that.

Note: This problem wouldn’t persist when the overriden method arguments are primitive data type. Because the compiler will select the most suitable method and executes it.

This article is contributed by **Nageswara Rao Maridu**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Constructors in Java

Constructors are used to initialize the object’s state. Like methods, a constructor also contains **collection of statements(i.e. instructions)** that are executed at time of Object creation.

**When is a Constructor called ?**
Each time an object is created using **new()** keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the **data members** of the same class.

Constructor is invoked at the time of object or instance creation. For Example:

```
class Geek
{
  .......

  // A Constructor
  new Geek() {}

  .......
}

// We can create an object of above class
// using below statement. This statement
// calls above constructor.
Geek obj = new Geek();
```

**Rules for writing Constructor:**

- Constructor(s) of a class must has same name as the class name in which it resides.
- A constructor in Java can not be abstract, final, static and Synchronized.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

**Types of constructor**

There are two type of constructor in Java:

1. **No-argument constructor:** A constructor that has no parameter is known as default constructor. If we don’t define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-argument then compiler does not create default constructor.
   Default constructor provides the default values to the object like 0, null etc. depending on the type.

   ```
   // Java Program to illustrate calling a
   // no-argument constructor
   import java.io.*;

   class Geek
   {
       int num;
       String name;

       // this would be invoked while object
       // of that class created.
       Geek()
       {
           System.out.println("Constructor called");
       }
   }

   class GFG
   {
       public static void main (String[] args)
       {
           // this would invoke default constructor.
           Geek geek1 = new Geek();

           // Default constructor provides the default
           // values to the object like 0, null
           System.out.println(geek1.name);
           System.out.println(geek1.num);
       }
   }
   ```

   Output :

   ```
   Constructor called
   null
   0
   ```

2. **Parameterized Constructor:** A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use parameterized constructor.

   ```
   // Java Program to illustrate calling of
   // parameterized constructor.
   import java.io.*;

   class Geek
   {
       // data members of the class.
       String name;
       int id;
   ```

```
    // contructor would initialized data members
    // with the values of passed arguments while
    // object of that class created.
    Geek(String name, int id)
    {
        this.name = name;
        this.id = id;
    }
}

class GFG
{
    public static void main (String[] args)
    {
        // this would invoke parameterized constructor.
        Geek geek1 = new Geek("adam", 1);
        System.out.println("GeekName :" + geek1.name +
                         " and GeekId :" + geek1.id);
    }
}
```

Output:

```
GeekName :adam and GeekId :1
```

**Does constructor return any value?**

There are no "return value" statements in constructor, but constructor returns current class instance. We can write 'return' inside a constructor.

**Constructor Overloading**

Like methods, we can overload constructors for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

```
// Java Program to illustrate constructor overloading
// using same task (addition operation ) for different
// types of arguments.
import java.io.*;

class Geek
{
    // constructor with one argument
    Geek(String name)
    {
        System.out.println("Constructor with one " +
                    "argument - String : " + name);
    }

    // constructor with two arguments
    Geek(String name, int age)
    {

        System.out.print("Constructor with two arguments : " +
                " String and Integer : " + name + " "+ age);

    }

    // Constructor with one argument but with different
    // type than previous..
    Geek(long id)
    {
        System.out.println("Constructor with one argument : " +
                                            "Long : " + id);
    }
}

class GFG
{
    public static void main(String[] args)
    {
        // Creating the objects of the class named 'Geek'
        // by passing different arguments

        // Invoke the constructor with one argument of
        // type 'String'.
        Geek geek2 = new Geek("Shikhar");

        // Invoke the constructor with two arguments
        Geek geek3 = new Geek("Dharmesh", 26);

        // Invoke the constructor with one argument of
        // type 'Long'.
        Geek geek4 = new Geek(325614567);
    }
}
```

Output:

```
Constructor with one argument - String : Shikhar
Constructor with two arguments - String and Integer : Dharmesh 26
Constructor with one argument - Long : 325614567
```

**How constructors are different from methods in Java?**

- Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
- Constructor(s) do not any return type while method(s) have the return type or **void** if does not return any value.
- Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

**Related Articles:**

- Constructor Chaining
- Copy constructor

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Default constructor in Java

Like C++, Java automatically creates default constructor if there is no default or parameterized constructor written by user, and (like C++) the default constructor automatically calls parent default constructor. But unlike C++, default constructor in Java initializes member data variable to default values (numeric values are initialized as 0, booleans are initialized as *false* and references are initialized as *null*).

For example, output of the below program is

```
0
null
false
0
0.0
```

```
// Main.java
class Test {
   int i;
   Test t;
   boolean b;
   byte bt;
   float ft;
}

public class Main {
    public static void main(String args[]) {
      Test t = new Test(); // default constructor is called.
      System.out.println(t.i);
      System.out.println(t.t);
      System.out.println(t.b);
      System.out.println(t.bt);
      System.out.println(t.ft);
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

References:
http://leepoint.net/notes-java/oop/constructors/constructor.html

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Assigning values to static final variables in Java

**Assigning values to static final variables in Java:**

In Java, non-static final variables can be assigned a value either in constructor or with the declaration. But, static final variables cannot be assigned value in constructor; they must be assigned a value with their declaration.

For example, following program works fine.

```
class Test {
  final int i;  // i could be assigned a value here also
  Tets() {
    i = 10;
  }

  //other stuff in the class
}
```

If we make *i* as *static final* then we must assign value to i with the delcaration.

```
class Test {
  static final int i = 10;  // Since i is static final, it must be assigned value here only.

  //other stuff in the class
}
```

Such behavior is obvious as static variables are shared among all the objects of a class; creating a new object would change the same static variable which is not allowed if the static variable is final.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Copy Constructor in Java

Prerequisite – Constructors in Java

Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

Following is an example Java program that shows a simple use of copy constructor.

```
// filename: Main.java

class Complex {

    private double re, im;

    // A normal parametrized constructor
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    // copy constructor
    Complex(Complex c) {
        System.out.println("Copy constructor called");
        re = c.re;
        im = c.im;
    }
    // Overriding the toString of Object class
    @Override
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);

        // Following involves a copy constructor call
        Complex c2 = new Complex(c1);

        // Note that following doesn't involve a copy constructor call as
        // non-primitive variables are just references.
        Complex c3 = c2;

        System.out.println(c2); // toString() of c2 is called here
    }
}
```

Output:

```
Copy constructor called
(10.0 + 15.0i)
```

Now try the following Java program:

```
// filename: Main.java

class Complex {

    private double re, im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
}

public class Main {

    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(c1);  // compiler error here
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Constructor Chaining In Java with Examples

Prerequisite – Constructors in Java

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Constructor chaining can be done in two ways:

- **Within same class**: It can be done using **this()** keyword for constructors in same class
- **From base class:** by using **super()** keyword to call constructor from the base class.

Constructor chaining occurs through **inheritance**. A sub class constructor's task is to call super class's constructor first. This ensures that creation of sub class's object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.

**Why do we need constructor chaining ?**

This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable.

**Constructor Chaining within same class using this() keyword :**

Constructor Chaining In Java

```
// Java program to illustrate Constructor Chaining
// within same class Using this() keyword
class Temp
```

```
{
    // default constructor 1
    // default constructor will call another constructor
    // using this keyword from same class
    Temp()
    {
        // calls constructor 2
        this(5);
        System.out.println("The Default constructor");
    }

    // parameterized constructor 2
    Temp(int x)
    {
        // calls constructor 3
        this(5, 15);
        System.out.println(x);
    }

    // parameterized constructor 3
    Temp(int x, int y)
    {
        System.out.println(x * y);
    }

    public static void main(String args[])
    {
        // invokes default constructor first
        new Temp();
    }
}
```

Output:

```
The Default constructor
5
75
```

**Rules of constructor chaining :**

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

**What happens if we change the order of constructors?**

Nothing, Constructor chaining can be achieved in any order

```
// Java program to illustrate Constructor Chaining
// within same class Using this() keyword
// and changing order of constructors
class Temp
{
    // default constructor 1
    Temp()
    {
        System.out.println("default");
    }

    // parameterized constructor 2
    Temp(int x)
    {
        // invokes default constructor
        this();
        System.out.println(x);
    }

    // parameterized constructor 3
    Temp(int x, int y)
    {
        // invokes parameterized constructor 2
        this(5);
        System.out.println(x * y);
    }

    public static void main(String args[])
    {
        // invokes parameterized constructor 3
        new Temp(8, 10);
    }
}
```

Output:

```
default
5
80
```

NOTE: In example 1, default constructor is invoked at the end, but in example 2 default constructor is invoked at first. Hence, order in constructor chaining is not important.

**Constructor Chaining to other class using super() keyword :**

```
// Java program to illustrate Constructor Chaining to
// other class using super() keyword
class Base
{
    String name;

    // constructor 1
    Base()
    {
        this("");
        System.out.println("No-argument constructor of" +
                                    " base class");
    }

    // constructor 2
    Base(String name)
    {
        this.name = name;
        System.out.println("Calling parameterized constructor"
                                        + " of base");
    }
}

class Derived extends Base
{
    // constructor 3
    Derived()
    {
        System.out.println("No-argument constructor " +
                        "of derived");
    }

    // parameterized constructor 4
    Derived(String name)
    {
        // invokes base class constructor 2
        super(name);
        System.out.println("Calling parameterized " +
                        "constructor of derived");
    }

    public static void main(String args[])
    {
        // calls parameterized constructor 4
        Derived obj = new Derived("test");

        // Calls No-argument constructor
        // Derived obj = new Derived();
    }
}
```

Output:

```
Calling parameterized constructor of base
Calling parameterized constructor of derived
```

Note : Similar to constructor chaining in same class, **super()** should be the first line of the constructor as super class's constructor are invoked before the sub class's constructor.

**Alternative method : using Init block** :
When we want certain common resources to be executed with every constructor we can put the code in the **init block**. Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

Example 1:

```
class Temp
{
    // block to be executed before any constructor.
    {
        System.out.println("init block");
    }

    // no-arg constructor
    Temp()
    {
        System.out.println("default");
    }

    // constructor with one arguemnt.
    Temp(int x)
    {
        System.out.println(x);
    }

    public static void main()
    {
        // Object creation by calling no-argument
        // constructor.
        new Temp();

        // Object creation by calling parameterized
        // constructor with one parameter.
        new Temp(10);
    }
}
```

Output:

```
init block
default
init block
10
```

NOTE: If there are more than one blocks, they are executed in the order in which they are are defined within the same class. See the ex.
Example :

```
class Temp
{
    // block to be excuted first
    {
        System.out.println("init");
    }
    Temp()
    {
        System.out.println("default");
    }
    Temp(int x)
    {
        System.out.println(x);
    }

    // block to be executed after the first block
    // which has been defined above.
    {
        System.out.println("second");
    }
    public static void main(String args[])
    {
        new Temp();
        new Temp(10);
    }
}
```

Output :

```
init
second
default
init
second
10
```

This article is contributed by **Apoorva singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Exceptions in Java

**What is an Exception?**

An exception is an event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.

When an error occurs in a method, the method creates an Object known as Exception Object and hands it off to the runTime system. The exception object contains information about the error, type of the error, and current state of the program when an error has occured. Creating the Exception Object and handling it to the runTime system is called throwing an Exception.

**Error vs Exception**

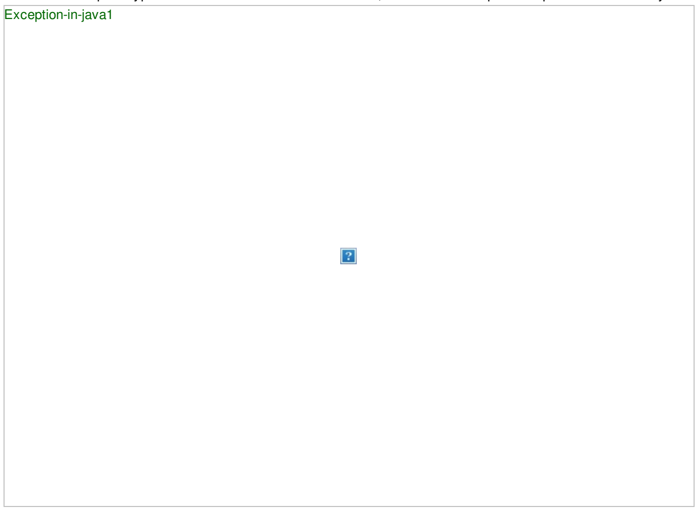
**Error:** An Error indicates serious problem that a reasonable application should not try to catch.
**Exception:** Exception indicates conditions that a reasonable application might try to catch.

**How to handle Exception?**

There might be the list of the methods that had been called to get to the method where error occured. This ordered list of the methods is called **Call Stack**.

- The runTime system searches the call stack to find the method that contains block of code that can handle the occured exception. The block of the code is called **Exception handler**.
- The runTime system starts searching from the method in which exception occured, proceeds through call stack in the reverse order in which methods were called.
- If it finds appropriate handler then it passes the occured exception to it. Appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
- If runTime system searches all the methods on call stack and couldn't have found the appropriate handler then the runTime system and consequently the running program terminates.

NOTE: All exception types are subclasses of class **Throwable**, which is at the top of exception class hierarchy.


Exception-in-java1

Example :

```
// Java program to demonstrate how exception is thrown.
class ThrowsExecp{

 public static void main(String args[]){

       String str = null;
       System.out.println(str.length());

 }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
 at ThrowsExecp.main(File.java:8)
```

An Example that illustrate how runTime system searches appropriate exception handler on the call stack :

```
// Java program to demonstrate exception is thrown
// how the runTime system searches th call stack
// to find appropriate exception handler.
class ExceptionThrown{

   // It throws the Exception.
   // Exception handler is not found within this method.
   static int divideByZero(int a, int b){

       if(b == 0){
           throw new ArithmeticException("Numerator should not be Zero.");
       }

       int i = a/b;
       return i;
```

```
    }

    // The runTime System searches the appropriate Exception handler
    // in this method also but couldn't have found. So looking forward
    // on the call stack.
    static int computeDivision(int a, int b) {

        int res =0;

        try
        {
          res = divideByZero(a,b);
        }
        catch(NumberFormatException ex)
        {
           System.out.println("NumberFormatException is occured.");
        }
        return res;
    }

    // In this method found appropriate Exception handler.
    // i.e. matching catch block.
  public static void main(String args[]){

        int a = 1;
        int b = 0;

        try
        {
            int i = computeDivision(a,b);

        }
        catch(ArithmeticException ex)
        {
            System.out.println(ex.getMessage());
        }
  }
}
```

Output :

```
Numerator should not be Zero.
```

See the below diagram to understand the flow of the call stack.

Related Articles:

- Types of Exceptions in Java
- Checked vs Unchecked Exceptions
- Throw- Throws in Java

Reference :
https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Exceptions

# Types of Exception in Java with Examples

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.
exceptions-in-java

**Built-in Exceptions**

Built-in exceptions are the exceptions which are available in Java libraries. These exceptions are suitable to explain certain error situations. Below is the list of important built-in exceptions in Java.

1. **Arithmetic Exception**
   It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundException**
   It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException**
   This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException**
   This Exception is raised when a file is not accessible or does not open.
5. **IOException**
   It is thrown when an input-output operation failed or interrupted
6. **InterruptedException**
   It is thrown when a thread is waiting , sleeping , or doing some processing , and it is interrupted.
7. **NoSuchFieldException**
   It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException**
   It is thrown when accessing a method which is not found.
9. **NullPointerException**

This exception is raised when referring to the members of a null object. Null represents nothing

10. **NumberFormatException**

    This exception is raised when a method could not convert a string into a numeric format.

11. **RuntimeException**

    This represents any exception which occurs during runtime.

12. **StringIndexOutOfBoundsException**

    It is thrown by String class methods to indicate that an index is either negative than the size of the string

**Examples of Built-in Exception:**

- **Arithmetic exception**

```
// Java program to demonstrate ArithmeticException
class ArithmeticException_Demo
{
    public static void main(String args[])
    {
        try {
            int a = 30, b = 0;
            int c = a/b;  // cannot divide by zero
            System.out.println ("Result = " + c);
        }
        catch(ArithmeticException e) {
            System.out.println ("Can't divide a number by 0");
        }
    }
}
```

**Output:**

```
Can't divide a number by 0
```

- **NullPointer Exception**

```
//Java program to demonstrate NullPointerException
class NullPointer_Demo
{
    public static void main(String args[])
    {
        try {
            String a = null; //null value
            System.out.println(a.charAt(0));
        } catch(NullPointerException e) {
            System.out.println("NullPointerException..");
        }
    }
}
```

**Output:**

```
NullPointerException..
```

- **StringIndexOutOfBound Exception**

```
// Java program to demonstrate StringIndexOutOfBoundsException
class StringIndexOutOfBound_Demo
{
    public static void main(String args[])
    {
        try {
            String a = "This is like chipping "; // length is 22
            char c = a.charAt(24); // accessing 25th element
            System.out.println(c);
        }
        catch(StringIndexOutOfBoundsException e) {
            System.out.println("StringIndexOutOfBoundsException");
        }
    }
}
```

**Output:**

```
StringIndexOutOfBoundsException
```

- **FileNotFound Exception**

```
//Java program to demonstrate FileNotFoundException
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
 class File_notFound_Demo {

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

            // Following file does not exist
            File file = new File("E://file.txt");

            FileReader fr = new FileReader(file);
        } catch (FileNotFoundException e) {
           System.out.println("File does not exist");
        }
    }
}
```

**Output:**

```
File does not exist
```

- **NumberFormat Exception**

```
// Java program to demonstrate NumberFormatException
class  NumberFormat_Demo
{
    public static void main(String args[])
    {
        try {
            // "akki" is not a number
            int num = Integer.parseInt ("akki") ;
            System.out.println(num);
        } catch(NumberFormatException e) {
            System.out.println("Number format exception");
        }
    }
}
```

**Output:**

```
Number format exception
```

- **ArrayIndexOutOfBounds Exception**

```
// Java program to demonstrate ArrayIndexOutOfBoundException
class ArrayIndexOutOfBound_Demo
{
    public static void main(String args[])
    {
        try{
            int a[] = new int[5];
            a[6] = 9; // accessing 7th element in an array of
                      // size 5
        }
        catch(ArrayIndexOutOfBoundsException e){
            System.out.println ("Array Index is Out Of Bounds");
        }
    }
}
```

**Output:**

```
Array Index is Out Of Bounds
```

**User-Defined Exceptions**

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, user can also create exceptions which are called ‘user-defined Exceptions’.
Following steps are followed for the creation of user-defined Exception.

- The user should create an exception class as a subclass of Exception class. Since all the exceptions are subclasses of Exception class, the user should also make his class a subclass of it. This is done as:

```
class MyException extends Exception
```

- We can write a default constructor in his own exception class.

```
MyException(){}
```

- We can also create a parameterized constructor with a string as a parameter.
  We can use this to store exception details. We can call super class(Exception) constructor from this and send the string there.

```
MyException(String str)
{
   super(str);
}
```

- To raise exception of user-defined type, we need to create an object to his exception class and throw it using throw clause, as:

```
MyException me = new MyException(“Exception details”);
throw me;
```

- The following program illustrates how to create own exception class MyException.
- Details of account numbers, customer names, and balance amounts are taken in the form of three arrays.
- In main() method, the details are displayed using a for-loop. At this time, check is done if in any account the balance amount is less than the minimum balance amount to be ept in the account.
- If it is so, then MyException is raised and a message is displayed “Balance amount is less”.

```
// Java program to demonstrate user defined exception

// This program throws an exception whenever balance
// amount is below Rs 1000
class MyException extends Exception
{
    //store account information
    private static int accno[] = {1001, 1002, 1003, 1004};

    private static String name[] =
            {"Nish", "Shubh", "Sush", "Abhi", "Akash"};

    private static double bal[] =
        {10000.00, 12000.00, 5600.0, 999.00, 1100.55};

    // default constructor
    MyException() {    }

    // parametrized constructor
    MyException(String str) { super(str); }

    // write main()
    public static void main(String[] args)
    {
        try  {
            // display the heading for the table
            System.out.println("ACCNO" + "\t" + "CUSTOMER" +
                                          "\t" + "BALANCE");

            // display the actual account information
            for (int i = 0; i < 5 ; i++)
            {
                System.out.println(accno[i] + "\t" + name[i] +
                                          "\t" + bal[i]);

                // display own exception if balance < 1000
                if (bal[i] < 1000)
                {
                    MyException me =
                       new MyException("Balance is less than 1000");
                    throw me;
                }
            }
        } //end of try
        catch (MyException e) {
            e.printStackTrace();
        }
    }
}
```

RunTime Error

```
MyException: Balance is less than 1000
at MyException.main(fileProperty.java:36)
```

**Output:**

```
ACCNO CUSTOMER BALANCE
1001 Nish 10000.0
1002 Shubh 12000.0
1003 Sush 5600.0
1004 Abhi 999.0
```

**Related Articles:**

- Checked vs Unchecked Exceptions in Java
- Catching base and derived classes as exceptions
- Quiz on Exception Handling

This article is contributed by **Nishant Sharma**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Exceptions

# Comparison of Exception Handling in C++ and Java

Both languages use *try*, *catch* and *throw* keywords for exception handling, and meaning of *try*, *catch* and *free* blocks is also same in both languages. Following are the differences between Java and C++ exception handling.

**1)** In C++, all types (including primitive and pointer) can be thrown as exception. But in Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception. For example, following type of code works in C++, but similar code doesn't work in Java.

```
#include <iostream>
using namespace std;
int main()
{
   int x = -1;

   // some other stuff
   try {
      // some other stuff
      if( x < 0 )
      {
         throw x;
      }
   }
   catch (int x ) {
      cout << "Exception occurred: thrown value is " << x << endl;
   }
   getchar();
   return 0;
}
```

Output:
*Exception occurred: thrown value is -1*

**2)** In C++, there is a special catch called "catch all" that can catch all kind of exceptions.

```
#include <iostream>
using namespace std;
int main()
{
   int x = -1;
   char *ptr;

   ptr = new char[256];

   // some other stuff
   try {
      // some other stuff
      if( x < 0 )
      {
         throw x;
      }
      if(ptr == NULL)
      {
         throw " ptr is NULL ";
      }
   }
   catch (...) // catch all
   {
      cout << "Exception occurred: exiting "<< endl;
      exit(0);
   }

   getchar();
   return 0;
}
```

Output:
*Exception occurred: exiting*

In Java, for all practical purposes, we can catch Exception object to catch all kind of exceptions. Because, normally we do not catch Throwable(s) other than Exception(s) (which are Errors)

```
catch(Exception e){
  …….
}
```

**3)** In Java, there is a block called *finally* that is always executed after the try-catch block. This block can be used to do cleanup work. There is no such block in C++.

```
// creating an exception type
class Test extends Exception { }

class Main {
   public static void main(String args[]) {

      try {
         throw new Test();
      }
      catch(Test t) {
         System.out.println("Got the Test Exception");
      }
      finally {
         System.out.println("Inside finally block ");
      }
  }
}
```

Output:
*Got the error*
*Inside finally block*

**4)** In C++, all exceptions are unchecked. In Java, there are two types of exceptions – checked and unchecked. See this for more details on checked vs Unchecked exceptions.

**5)** In Java, a new keyword *throws* is used to list exceptions that can be thrown by a function. In C++, there is no *throws* keyword, the same keyword *throw* is used for this purpose also.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

C/C++ Puzzles
Java
cpp-exception
Java-Exceptions

# Catching base and derived classes as exceptions

**Exception Handling – catching base and derived classes as exceptions:**

If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class.

If we put base class first then the derived class catch block will never be reached. For example, following C++ code prints *"Caught Base Exception"*

```
#include<iostream>
using namespace std;

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   // some other stuff
   try {
       // Some monitored code
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   catch(Derived d) {  //This catch block is NEVER executed
        cout<<"Caught Derived Exception";
   }
   getchar();
   return 0;
}
```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable. Following is the modifed program and it prints *"Caught Derived Exception"*

```
#include<iostream>
using namespace std;

class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   // some other stuff
   try {
       // Some monitored code
       throw d;
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   getchar();
   return 0;
}
```

In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.
For example, following Java code fails in compilation with error message *"exception Derived has already been caught"*

```
//filename Main.java
class Base extends Exception {}
class Derived extends Base  {}
public class Main {
  public static void main(String args[]) {
    try {
        throw new Derived();
    }
    catch(Base b) {}
    catch(Derived d) {}
  }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

C/C++ Puzzles
CPP

# Checked vs Unchecked Exceptions in Java

In Java, there two types of exceptions:

**1) Checked:** are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using *throws* keyword.

For example, consider the following Java program that opens file at locatiobn "C:\test\a.txt" and prints first three lines of it. The program doesn't compile, because the function main() uses FileReader() and FileReader() throws a checked exception *FileNotFoundException*. It also uses readLine() and close() methods, and these methods also throw checked exception *IOException*

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Output:

```
Exception in thread "main" java.lang.RuntimeException: Uncompilable source code -
unreported exception java.io.FileNotFoundException; must be caught or declared to be
thrown
 at Main.main(Main.java:5)
```

To fix the above program, we either need to specify list of exceptions using throws, or we need to use try-catch block. We have used throws in the below program. Since *FileNotFoundException* is a subclass of *IOException*, we can just specify *IOException* in the throws list and make the above program compiler-error-free.

```
import java.io.*;

class Main {
    public static void main(String[] args) throws IOException {
        FileReader file = new FileReader("C:\\test\\a.txt");
        BufferedReader fileInput = new BufferedReader(file);

        // Print first 3 lines of file "C:\test\a.txt"
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }
}
```

Output: First three lines of file "C:\test\a.txt"

**2) Unchecked** are the exceptions that are not checked at compiled time. In C++, all exceptions are unchecked, so it is not forced by the compiler to either handle or specify the exception. It is up to the programmers to be civilized, and specify or catch the exceptions.
In Java exceptions under *Error* and *RuntimeException* classes are unchecked exceptions, everything else under throwable is checked.

```
                   +-----------+
    | Throwable |
                   +-----------+
                    /        \
    /          \
         +-------+          +-----------+
         | Error |          | Exception |
         +-------+          +-----------+
   /  |  \         / |       \
        \_________/  \_______/      \
                   +------------------+
 unchecked  checked | RuntimeException |
    +------------------+
      /   |    |     \
     \_____________________/

       unchecked
```

Consider the following Java program. It compiles fine, but it throws *ArithmeticException* when run. The compiler allows it to compile, because *ArithmeticException* is an unchecked exception.

```
class Main {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```

Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at Main.main(Main.java:5)
Java Result: 1
```

**Why two types?**
See Unchecked Exceptions — The Controversy for details.

**Should we make our exceptions checked or unchecked?**

Following is the bottom line from Java documents

*If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception*

**Java Corner on GeeksforGeeks**

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Exceptions

# throw and throws in Java

**throw:**

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. General form of **throw** is as shown below:

```
throw Instance
```

But this exception i.e, *Instance* must be of type **Throwable** or a subclass of **Throwable**. For example Exception is a sub-class of Throwable and user defined exceptions typically extend Exception class. Unlike C++, data types such as int, char, floats or non-throwable classes cannot be used as exceptions.

The flow of execution of the program stops immediately after the throw statement is executed and the nearest enclosing **try** block is checked to see if it has a **catch** statement that matches the type of exception. If it finds a match, controlled is transferred to that statement otherwise next enclosing **try** block is checked and so on. If no matching **catch** is found then the default exception handler will halt the program.

The below program explains the use of **throw**:

```
// Java program that demonstrates the use of throw
class ThrowExcep
{
    static void fun()
    {
        try
        {
            throw new NullPointerException("demo");
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught inside fun().");
            throw e; // rethrowing the exception
        }
    }

    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(NullPointerException e)
        {
            System.out.println("Caught in main.");
        }
    }
}
```

Output:

```
Caught inside fun().
Caught in main.
```

**throws:**

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.
Below is the general form of a method which includes a **throws** clause:

```
type method_name(parameters) throws exception_list
```

Here, the exception_list is a comma separated list of all the exceptions which a method might throw. Below is a Java program to illustrate the use of **throws**:

```
// Java program to demonstrate working of throws
class ThrowsExecp
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException("demo");
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalAccessException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

Output:

```
Inside fun().
caught in main.
```

**Reference:**
Java – The complete Reference by Herbert Schildt

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## User-defined Custom Exception in Java

Java provides us facility to create our own exceptions which are basically derived classes of Exception. For example MyException in below code extends the Exception class.

We pass the string to the constructor of the super class- Exception which is obtained using "getMessage()" function on the object created.

```
// A Class that represents use-defined expception
class MyException extends Exception
{
    public MyException(String s)
    {
        // Call constructor of parent Exception
        super(s);
    }
}

// A Class that uses above MyException
public class Main
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException("GeeksGeeks");
        }
        catch (MyException ex)
        {
            System.out.println("Caught");

            // Print the message from MyException object
            System.out.println(ex.getMessage());
        }
    }
}
```

Output:

```
Caught
GeeksGeeks
```

In the above code, constructor of MyException requires a string as its argument. The string is passed to parent class Exception's constructor using super(). The constructor of Exception class can also be called without a parameter and call to super is not mandatory.

```
// A Class that represents use-defined expception
class MyException extends Exception
{

}

// A Class that uses above MyException
public class setText
{
    // Driver Program
    public static void main(String args[])
    {
        try
        {
            // Throw an object of user defined exception
            throw new MyException();
        }
        catch (MyException ex)
        {
            System.out.println("Caught");
            System.out.println(ex.getMessage());
        }
    }
}
```

Output:

```
Caught
null
```

This article is contributed by **Pranjal Mathur**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## Infinity or Exception in Java when divide by 0?

Consider the following code snippets:

```
public class Geeksforgeeks
{
    public static void main(String[] args)
    {
        double p = 1;
        System.out.println(p/0);
    }
}
```

**Output**:

```
Infinity
```

```
public class Geeksforgeeks
{
    public static void main(String[] args)
    {
        int p = 1;
        System.out.println(p/0);
    }
}
```

**Output:**

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at Geeksforgeeks.main(Geeksforgeeks.java:8)
```

**Explanation**: In the first piece of code, a double value is being divided by 0 while in the other case an integer value is being divide by 0. However the solution for both of them differs.

- In case of double/float division, the output is **Infinity**, the basic reason behind that it implements the floating point arithmetic algorithm which specifies a special values like “Not a number” OR “infinity” for “divided by zero cases” as per IEEE 754 standards.
- In case of integer division, it throws ArithmeticException.

This article is contributed by **Pranjal Mathur**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Exception Handling
Java

# Multicatch in Java

**Background**

Prior to Java 7, we had to catch only one exception type in each catch block. So whenever we needed to handle more than one specific exception, but take same action for all exceptions, then we had to have more than one catch block containing the same code.
In the following code, we have to handle two different exceptions but take same action for both. So we needed to have two different catch blocks as of Java 6.0.

```
// A Java program to demonstrate that we needed
// multiple catch blocks for multiple exceptions
// prior to Java 7
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
        catch (ArithmeticException ex)
        {
            System.out.println("Arithmetic " + ex);
        }
        catch (NumberFormatException ex)
        {
            System.out.println("Number Format Exception " + ex);
        }
    }
}
```

**Input 1:**

```
GeeksforGeeks
```

**Output 2:**

```
Exception encountered java.lang.NumberFormatException:
For input string: "GeeksforGeeks"
```

**Input 2:**

```
0
```

**Output 2:**

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

**Multicatch**

Starting from Java 7.0, it is possible for a single catch block to catch multiple exceptions by separating each with | (pipe symbol) in catch block.

```
// A Java program to demonstrate multicatch
// feature
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        Scanner scn = new Scanner(System.in);
        try
        {
            int n = Integer.parseInt(scn.nextLine());
            if (99%n == 0)
                System.out.println(n + " is a factor of 99");
        }
        catch (NumberFormatException | ArithmeticException ex)
        {
            System.out.println("Exception encountered " + ex);
        }
    }
}
```

**Input 1:**

```
GeeksforGeeks
```

**Output 1:**

```
Exception encountered java.lang.NumberFormatException:
For input string: "GeeksforGeeks"
```

**Input 2:**

```
0
```

**Output 2:**

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

A catch block that handles multiple exception types creates no duplication in the bytecode generated by the compiler, that is, the bytecode has no replication of exception handlers.

**Important Points:**

- If all the exceptions belong to the same class hierarchy, we should catching the base exception type. However, to catch each exception, it needs to be done separately in their own catch blocks.
- Single catch block can handle more than one type of exception. However, the base (or ancestor) class and subclass (or descendant) exceptions can not be caught in one statement. For Example

  ```
  // Not Valid as Exception is an ancestor of
  // NumberFormatException
  catch(NumberFormatException | Exception ex)
  ```

- All the exceptions must be separated by vertical bar pipe |.

This article is contributed by **Aparna Vadlamani**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Exceptions

# Chained Exceptions in Java

Chained Exceptions allows to relate one exception with another exception, i.e one exception describes cause of another exception. For example, consider a situation in which a method throws an ArithmeticException because of an attempt to divide by zero but the actual cause of exception was an I/O error which caused the divisor to be zero. The method will throw only ArithmeticException to the caller. So the caller would not come to know about the actual cause of exception. Chained Exception is used in such type of situations.

**Constructors** Of Throwable class Which support chained exceptions in java :

1. Throwable(Throwable cause) :- Where cause is the exception that causes the current exception.
2. Throwable(String msg, Throwable cause) :- Where msg is the exception message and cause is the exception that causes the current exception.

**Methods** Of Throwable class Which support chained exceptions in java :

1. getCause() method :- This method returns actual cause of an exception.
2. initCause(Throwable cause) method :- This method sets the cause for the calling exception.

Example of using Chained Exception:

```
// Java program to demonstrate working of chained exceptions
public class ExceptionHandling
{
    public static void main(String[] args)
    {
        try
        {
            // Creating an exception
            NumberFormatException ex =
                       new NumberFormatException("Exception");

            // Setting a cause of the exception
            ex.initCause(new NullPointerException(
                      "This is actual cause of the exception"));

            // Throwing an exception with cause.
            throw ex;
        }

        catch(NumberFormatException ex)
        {
            // displaying the exception
            System.out.println(ex);

            // Getting the actual cause of the exception
            System.out.println(ex.getCause());
        }
    }
}
```

Output:

```
java.lang.NumberFormatException: Exception
java.lang.NullPointerException: This is actual cause of the exception
```

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Regular Expressions in Java

Regular Expressions or Regex (in short) is an API for defining String patterns that can be used for searching, manipulating and editing a text. It is widely used to define constraint on strings such as password. Regular Expressions are provided under java.util.regex package.

**The java.util.regex** package primarily consists of the following three classes:

1. **util.regex.Pattern** – Used for defining patterns
2. **util.regex.Matcher** – Used for performing match operations on text using patterns
3. **PatternSyntaxException**– Used for indicating syntax error in a regular expression pattern

**java.util.regex.Pattern** Class

1. **matches()**– It is used to check if the whole text matches a pattern. Its output is boolean.

```
// A Simple Java program to demonstrate working of
// Pattern.matches() in Java
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Following line prints "true" because the whole
        // text "geeksforgeeks" matches pattern "geeksforge*ks"
        System.out.println (Pattern.matches("geeksforge*ks",
                                    "geeksforgeeks"));

        // Following line prints "false" because the whole
        // text "geeksfor" doesn't match pattern "g*geeks*"
        System.out.println (Pattern.matches("g*geeks*",
                                    "geeksfor"));
    }
}
```

Output:

```
true
false
```

2. **compile()**– Used to create a pattern object by compiling a given string that may contain regular expressions. Input may also contains flags like Pattern.CASE_INSENSITIVE, Pattern.COMMENTS, .. etc (See this for details).
3. **split()**– It is used to split a text into multiple strings based on a delimiter pattern.

**java.util.regex.Matcher** Class

1. **find()** –It is mainly used for searching multiple occurrences of the regular expressions in the text.
2. **start()** – It is used for getting the start index of a match that is being found using find() method.
3. **end()** –It is used for getting the end index of a match that is being found using find() method. It returns index of character next to last matching character

Note that Pattern.matches() checks if whole text matches with a pattern or not. Other methods (demonstrated below) are mainly used to find multiple occurrences of pattern in text.

**Java Programs to demonstrate workings of compile(), find(), start(), end() and split() :**

1. **Java Program to demonstrate simple pattern searching**

```
// A Simple Java program to demonstrate working of
// String matching in Java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Create a pattern to be searched
        Pattern pattern = Pattern.compile("geeks");

        // Search above pattern in "geeksforgeeks.org"
        Matcher m = pattern.matcher("geeksforgeeks.org");

        // Print starting and ending indexes of the pattern
        // in text
        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output:

```
Pattern found from 0 to 4
Pattern found from 8 to 12
```

2. **Java Program to demonstrate simple regular expression searching**

```
// A Simple Java program to demonstrate working of
// String matching in Java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Create a pattern to be searched
        Pattern pattern = Pattern.compile("ge*");

        // Search above pattern in "geeksforgeeks.org"
        Matcher m = pattern.matcher("geeksforgeeks.org");

        // Print starting and ending indexes of the pattern
        // in text
        while (m.find())
```

```
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output:

```
Pattern found from 0 to 2
Pattern found from 8 to 10
Pattern found from 16 to 16
```

3. **Java program to demonstrate Case Insensitive Searching**

```
// A Simple Java program to demonstrate working of
// String matching in Java
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        // Create a pattern to be searched
        Pattern pattern = Pattern.compile("ge*", Pattern.CASE_INSENSITIVE);

        // Search above pattern in "geeksforgeeks.org"
        Matcher m = pattern.matcher("GeeksforGeeks.org");

        // Print starting and ending indexes of the pattern
        // in text
        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output:

```
Pattern found from 0 to 2
Pattern found from 8 to 10
Pattern found from 16 to 16
```

4. **Java program to demonstrate working of split() to split a text based on a delimiter pattern**

```
// Java program to demonstrate working of splitting a text by a
// given pattern
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Demo
{
    public static void main(String args[])
    {
        String text = "geeks1for2geeks3";

        // Specifies the string pattern which is to be searched
        String delimiter =  "\\d";
        Pattern pattern = Pattern.compile(delimiter,
                                Pattern.CASE_INSENSITIVE);

        // Used to perform case insensitive search of the string
        String[] result = pattern.split(text);


        for (String temp: result)
            System.out.println(temp);
    }
}
```

Output:

```
geeks
for
geeks
```

**Important Observations/Facts:**

1. We create a pattern object by calling Pattern.compile(), there is no constructor. compile() is a static method in Pattern class.
2. Like above, we create a Matcher object using matcher() on objects of Pattern class.
3. Pattern.matches() is also a static method that is used to check if given text as a whole matches pattern or not.
4. find() is used to find multiple occurrences of pattern in text.
5. We can split a text based on a delimiter pattern using split()

Quantifiers in Java

This article is contributed by **Akash Ojha**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Java
Technical Scripter
Java

## Quantifiers in Java

We strongly recommend to refer below post as a prerequisite of this.

Regular Expressions in Java

Quantifiers allow user to specify the number of occurrences to match against. Below are some commonly used quantifiers in Java.

```
X*      Zero or more occurrences of X
X?      Zero or One occurrences of X
X+      One or More occurrences of X
X{n}    Exactly n occurrences of X
```

```
X{n, }    At-least n occurrences of X
X{n, m}   Count of occurrences of X is from n to m
```

The above quantifiers can be made Greedy, Reluctant and Possessive.

**Greedy quantifier** (Default)
By default, quantifiers are Greedy. Greedy quantifiers try to match the longest text that matches given pattern. Greedy quantifiers work by first reading the entire string before trying any match. If the entire text doesn't match, remove last character and try again, repeating the process until a match is found.

```
// Java program to demonstrate Greedy Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class
        // By default quantifier "+" is Greedy
        Pattern p = Pattern.compile("g+");

        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");

        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));

    }
}
```

Output :

```
Pattern found from 0 to 2
```

**Explanation :** The pattern **g+** means one or more occurrences of **g**. Text is **ggg**. The greedy matcher would match the longest text even if parts of matching text also match. In this example, **g** and **gg** also match, but the greedy matcher produces **ggg**.

**Reluctant quantifier** (Appending a **?** after quantifier)
This quantifier uses the approach that is opposite of greedy quantifiers. It starts from first character and processes one character at a time.

```
// Java program to demonstrate Reluctant Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class
        // Here "+" is a Reluctant quantifier because
        // a "?' is appended after it.
        Pattern p = Pattern.compile("g+?");

        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");

        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));

    }
}
```

Output :

```
Pattern found from 0 to 0
Pattern found from 1 to 1
Pattern found from 2 to 2
```

**Explanation :** Since the quantifier is reluctant, it matches the shortest part of test with pattern. It processes one character at a time.

**Possessive quantifier** (Appending a **+** after quantifier)
This quantifier matches as many characters as it can like greedy quantifier. But if the entire string doesn't match, then it doesn't try removing characters from end.

```
// Java program to demonstrate Possessive Quantifiers
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Making an instance of Pattern class
        // Here "+" is a Possessive quantifier because
        // a "+' is appended after it.
        Pattern p = Pattern.compile("g++");

        // Making an instance of Matcher class
        Matcher m = p.matcher("ggg");

        while (m.find())
            System.out.println("Pattern found from " + m.start() +
                               " to " + (m.end()-1));
    }
}
```

Output :

```
Pattern found from 0 to 2
```

Explanation: We get the same output as Greedy because whole text matches the pattern.

**Below is an example to show difference between Greedy and Possessive Quantifiers.**

```
// Java program to demonstrate difference between Possessive and
// Greedy Quantifiers
```

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

class Test
{
    public static void main(String[] args)
    {
        // Create a pattern with Greedy quantifier
        Pattern pg = Pattern.compile("g+g");

        // Create same pattern with possessive quantifier
        Pattern pp = Pattern.compile("g++g");

        System.out.println("Using Greedy Quantifier");
        Matcher mg = pg.matcher("ggg");
        while (mg.find())
            System.out.println("Pattern found from " + mg.start() +
                               " to " + (mg.end()-1));

        System.out.println("\nUsing Possessive Quantifier");
        Matcher mp = pp.matcher("ggg");
        while (mp.find())
            System.out.println("Pattern found from " + mp.start() +
                               " to " + (mp.end()-1));

    }
}
```

Output :

```
Using Greedy Quantifier
Pattern found from 0 to 2

Using Possessive Quantifier
```

In the above example, since first quantifier is greedy, **g+** matches with whole string. If we match **g+** with whole string, **g+g** doesn't match, the Greedy quantifier, removes last character, matches **gg** with **g+** and finds a match.
In Possessive quantifier, we start like Greedy. **g+** matches whole string, but matching **g+** with whole string doesn't match **g+g** with **ggg**. Unlike Greedy, since quantifier is possessive, we stop at this point.

This article is contributed by **Rahul Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Technical Scripter

# Object class in Java

**Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

**Using Object class methods**

There are 12 methods in **Object** class:

- **toString()** : toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, it is defined as:

  ```
  // Default behavior of toString() is to print class name, then
  // @, then unsigned hexadecimal representation of the hash code
  // of the object
  public String toString()
  {
      return getClass().getName() + "@" + Integer.toHexString(hashCode());
  }
  ```

  It is always recommended to override **toString()** method to get our own String representation of Object. For more on override of toString() method refer – Overriding toString() in Java
  **Note :** Whenever we try to print any Object reference, then internally toString() method is called.

  ```
  Student s = new Student();

  // Below two statements are equivalent
  System.out.println(s);
  System.out.println(s.toString());
  ```

- **hashCode()** : For every object, JVM generates a unique number which is hashcode. It returns distinct integers for distinct objects. A common misconception about this method is that hashCode() method returns the address of object, which is not correct. It convert the internal address of object to an integer by using an algorithm. The hashCode() method is **native** because in Java it is impossible to find address of an object, so it uses native languages like C/C++ to find address of the object.
  **Use of hashCode() method :** Returns a hash value that is used to search object in a collection. JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.
  **Note :** Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example,for a Student class we can return roll no. of student from hashCode() method as it is unique.

  ```
  // Java program to demonstrate working of
  // hasCode() and toString()
  public class Student
  {
      static int last_roll = 100;
      int roll_no;

      // Constructor
      Student()
      {
          roll_no = last_roll;
          last_roll++;
      }

      // Overriding hashCode()
      @Override
      public int hashCode()
      {
          return roll_no;
      }

      // Driver code
  ```

```
    public static void main(String args[])
    {
        Student s = new Student();

        // Below two statements are equivalent
        System.out.println(s);
        System.out.println(s.toString());
    }
}
```

Output :

```
Student@64
Student@64
```

Note that $4*16^0 + 6*16^1 = 100$

- **equals(Object obj)** : Compares the given object to "this" object (the object on which the method is called). It gives a generic way to compare objects for equality. It is recommended to override **equals(Object obj)** method to get our own equality condition on Objects. For more on override of equals(Object obj) method refer – Overriding equals method in Java
  **Note :** It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.
- **getClass()** : Returns the class object of "this" object and used to get actual runtime class of the object. It can also be used to get metadata of this class. The returned Class object is the object that is locked by static synchronized methods of the represented class. As it is final so we don't override it.

```
// Java program to demonstrate working of getClass()
public class Test
{
    public static void main(String[] args)
    {
        Object obj = new String("GeeksForGeeks");
        Class c = obj.getClass();
        System.out.println("Class of Object obj is : "
                          + c.getName());
    }
}
```

Output:

```
Class of Object obj is : java.lang.String
```

**Note :**After loading a .class file, JVM will create an object of the type *java.lang.Class* in the Heap area. We can use this class object to get Class level information. It is widely used in Reflection

- **finalize()** method : This method is called just before an object is garbage collected. It is called by the Garbage Collector on an object when garbage collector determines that there are no more references to the object. We should override finalize() method to dispose system resources, perform clean-up activities and minimize memory leaks. For example before destroying Servlet objects web container, always called finalize method to perform clean-up activities of the session.
  **Note :**finalize method is called just **once** on an object even though that object is eligible for garbage collection multiple times.

```
// Java program to demonstrate working of finalize()
public class Test
{
    public static void main(String[] args)
    {
        Test t = new Test();
        System.out.println(t.hashCode());

        t = null;

        // calling garbage collector
        System.gc();

        System.out.println("end");
    }

    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

Output:

```
366712642
end
finalize method called
```

- **clone()** : It returns a new object that is exactly the same as this object. For clone() method refer Clone()
- The remaining three methods **wait()**, **notify() notifyAll()** are related to Concurrency. Refer Inter-thread Communication in Java for details.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

## Java and Multiple Inheritance

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

**Why Java doesn't support Multiple Inheritance?**

Consider the below Java code. It shows error.

```
// First Parent class
class Parent1
{
    void fun()
    {
        System.out.println("Parent1");
    }
}
```

```
// Second Parent Class
class Parent2
{
    void fun()
    {
        System.out.println("Parent2");
    }
}

// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

Output :

```
Compiler Error
```

From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method.

**1. The Diamond Problem:**

```
          GrandParent
           /     \
          /       \
      Parent1      Parent2
          \       /
           \     /
             Test
```

```
// A Grand parent class in diamond
class GrandParent
{
    void fun()
    {
        System.out.println("Grandparent");
    }
}

// First Parent class
class Parent1 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
class Parent2 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent2");
    }
}


// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Beta's fun() method.

Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

**2. Simplicity –** Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.

**How are above problems handled for Default Methods and Interfaces ?**

Java 8 supports default methods where interfaces can provide default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

```
// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface PI1
{
    // default method
    default void show()
    {
        System.out.println("Default PI1");
    }
}

interface PI2
{
    // Default method
    default void show()
    {
        System.out.println("Default PI2");
    }
}

// Implementation class code
class TestClass implements PI1, PI2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // use super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

```
Default PI1
Default PI2
```

If we remove implementation of default method from “TestClass”, we get compiler error. See this for a sample run.

If there is a diamond through interfaces, then there is no issue if none of the middle interfaces provide implementation of root interface. If they provide implementation, then implementation can be accessed as above using super keyword.

```
// A simple Java program to demonstrate how diamond
// problem is handled in case of default methods

interface GPI
{
    // default method
    default void show()
    {
        System.out.println("Default GPI");
    }
}

interface PI1 extends GPI { }

interface PI2 extends GPI { }

// Implementation class code
class TestClass implements PI1, PI2
{
    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

```
Default GPI
```

Also See – http://qa.geeksforgeeks.org/510/why-java-doesnt-support-multiple-inheritance

This article is contributed by **Vishal S**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Java Object Creation of Inherited Class

In inheritance, subclass acquires super class properties. An important point to note is, when subclass object is created, a separate object of super class object will not be created. Only a subclass object object is created that has super class variables.

This situation is different from a normal assumption that a constructor call means an object of the class is created, so we can’t blindly say that whenever a class constructor is executed, object of that class is created or not.

```
// A Java program to demonstrate that both super class
// and subclass constructors refer to same object

// super class
class Fruit
{
    public Fruit()
    {
        System.out.println("Super class constructor");
        System.out.println("Super class object hashcode :" +
                        this.hashCode());
        System.out.println(this.getClass().getName());
    }
}

// sub class
class Apple extends Fruit
{
    public Apple()
    {
        System.out.println("Subclass constructor invoked");
        System.out.println("Sub class object hashcode :" +
                        this.hashCode());
        System.out.println(this.hashCode() + "   " +
                        super.hashCode());

        System.out.println(this.getClass().getName() + "  " +
                        super.getClass().getName());
    }
}
```

```
// driver class
public class Test
{
    public static void main(String[] args)
    {
        Apple myApple = new Apple();
    }
}
```

Output:

```
super class constructor
super class object hashcode :366712642
Apple
sub class constructor
sub class object hashcode :366712642
366712642   366712642
Apple  Apple
```

As we can see that both super class(Fruit) object hashcode and subclass(Apple) object hashcode are same, so only one object is created. This object is of class Apple(subclass) as when we try to print name of class which object is created, it is printing Apple which is subclass.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Comparison of Inheritance in C++ and Java

The purpose of inheritance is same in C++ and Java. Inheritance is used in both languages for reusing code and/or creating is-a relationship. There are following differences in the way both languages provide support for inheritance.

**1)** In Java, all classes inherit from the Object class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and Object class is root of the tree. In Java, if we create a class that doesn't inherit from any class then it automatically inherits from Object class . In C++, there is forest of classes; when we create a class that doesn't inherit from anything, we create a new tree in forest.

Following Java example shows that Test class automatically inherits from Object class.

```
class Test {
    // members of test
}
class Main {
  public static void main(String[] args) {
    Test t = new Test();
    System.out.println("t is instanceof Object: " + (t instanceof Object));
  }
}
```

Output:

```
t is instanceof Object: true
```

**2)** In Java, members of the grandparent class are not directly accessible. See this G-Fact for more details.

**3)** The meaning of protected member access specifier is somewhat different in Java. In Java, protected members of a class "A" are accessible in other class "B" of same package, even if B doesn't inherit from A (they both have to be in the same package). For example, in the following program, protected members of A are accessible in B.

```
// filename B.java
class A {
    protected int x = 10, y = 20;
}

class B {
    public static void main(String args[]) {
        A a = new A();
        System.out.println(a.x + " " + a.y);
    }
}
```

**4)** Java uses *extends* keyword for inheritence. Unlike C++, Java doesn't provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class. Like C++, private members of base class are not accessible in derived class.
Unlike C++, in Java, we don't have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.

**5)** In Java, methods are virtual by default. In C++, we explicitly use virtual keyword. See this G-Fact for more details.

**6)** Java uses a separte keyword *interface* for interfaces, and *abstract* keyword for abstract classes and abstract functions.

Following is a Java abstract class example.

```
// An abstract class example
abstract class myAbstractClass {

   // An abstract method
   abstract void myAbstractFun();

   // A normal method
   void fun() {
      System.out.println("Inside My fun");
   }
}

public class myClass extends myAbstractClass {
   public void myAbstractFun() {
      System.out.println("Inside My fun");
   }
}
```

Following is a Java interface example

```
// An interface example
public interface myInterface {
   // myAbstractFun() is public and abstract, even if we don't use these keywords
   void myAbstractFun();  // is same as public abstract void myAbstractFun()
}

// Note the implements keyword also.
public class myClass implements myInterface {
   public void myAbstractFun() {
      System.out.println("Inside My fun");
   }
}
```

**7)** Unlike C++, Java doesn't support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.

**8 )** In C++, default constructor of parent class is automatically called, but if we want to call parametrized constructor of a parent class, we must use Initializer list. Like C++, default constructor of the parent class is automatically called in Java, but if we want to call parametrized constructor then we must use super to call the parent constructor. See following Java example.

```
package main;

class Base {
    private int b;
    Base(int x) {
        b = x;
        System.out.println("Base constructor called");
    }
}

class Derived extends Base {
    private int d;
    Derived(int x, int y) {
        // Calling parent class parameterized constructor
        // Call to parent constructor must be the first line in a Derived class
        super(x);
        d = y;
        System.out.println("Derived constructor called");
    }
}

class Main{
    public static void main(String[] args) {
      Derived obj = new Derived(1, 2);
    }
}
```

Output:

```
Base constructor called
Derived constructor called
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Inheritance and constructors in Java

In Java, constructor of base class with no argument gets automatically called in derived class constructor. For example, output of following program is:

*Base Class Constructor Called*
*Derived Class Constructor Called*

```
// filename: Main.java
class Base {
  Base() {
    System.out.println("Base Class Constructor Called ");
  }
}

class Derived extends Base {
  Derived() {
    System.out.println("Derived Class Constructor Called ");
  }
}

public class Main {
  public static void main(String[] args) {
    Derived d = new Derived();
  }
}
```

But, if we want to call parameterized contructor of base class, then we can call it using super(). The point to note is **base class comstructor call must be the first line in derived class constructor**. For example, in the following program, super(_x) is first line derived class constructor.

```
// filename: Main.java
class Base {
  int x;
  Base(int _x) {
    x = _x;
  }
}

class Derived extends Base {
  int y;
  Derived(int _x, int _y) {
    super(_x);
    y = _y;
  }
  void Display() {
    System.out.println("x = "+x+", y = "+y);
  }
}

public class Main {
    public static void main(String[] args) {
        Derived d = new Derived(10, 20);
        d.Display();
    }
}
```

Output:

*x = 10, y = 20*

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# Overriding in Java

Child class can redefine methods of the parent class. This is called function overriding. The signature (return type, parameter type and number of parameters) is kept same as defined in the parent class. Method overriding is done to achieve Run Time Polymorphism (An overridden method is called according to the object invoking it, not according to the reference type).

```
// A Simple Java program to demonstrate overriding
// in Java

// Base Class
class Parent
{
    void show() { System.out.println("Parent's show()"); }
}

// Inherited class
class Child extends Parent
{
    // This method overrides show() of Parent
    void show() { System.out.println("Child's show()"); }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();

        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
        Parent obj2 = new Child();
        obj2.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

**Some Interesting Facts:**

**1)** In C++, we need virtual keyword to achieve overriding or Run Time Polymorphism. In Java, methods are virtual by default.

**2)** We can have multilevel overriding.

```
// A Simple Java program to demonstrate multi-level
// overriding in Java

// Base Class
class Parent
{
    void show() { System.out.println("Parent's show()"); }
}

// Inherited class
class Child extends Parent
{
    // This method overrides show() of Parent
    void show() { System.out.println("Child's show()"); }
}

// Inherited class
class GrandChild extends Child
{
    // This method overrides show() of Parent
    void show() { System.out.println("GrandChild's show()"); }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        Parent obj1 = new GrandChild();
        obj1.show();
    }
}
```

Output :

```
GrandChild's show()
```

**3)** If we don't want a method to be overridden, we declare it as **final**.

```
// A Simple Java program to demonstrate working
// of final and overriding in Java

class Parent
{
    // Can't be overridden
    final void show() {  }
}

class Child extends Parent
{
    // This would produce error
    void show() {  }
}
```

Output :

```
13: error: show() in Child cannot override show() in Parent
    void show() {  }
         ^
  overridden method is final
```

**4)** In overridden method, we can not give weaker access, but reverse is possible. For example, a protected method in base class cannot be overridden as private in derived class.

```
// This program won't compile as we have
// given weaker access in derived class
class Parent
{
    // Can't be overridden
    protected void show() {  }
}

class Child extends Parent
{
    // This would produce error. If we
    // replace private with protected or
    // public, the program would work.
    private void show() {  }
}
```

Output :

```
12: error: show() in Child cannot override show() in Parent
    private void show() {  }
                 ^
  attempting to assign weaker access privileges; was protected
1 error
```

**5)** Private methods cannot be overridden as they are bonded during compile time. Therefore we can't even override private methods in an inner class (See this for details)

**6)** We can declare static methods with same signature in subclass, but it is not considered overriding as there won't be any run-time polymorphism. Hence the answer is 'No'. Please see this for details.

**7)** We can call parent class method in overridden method using super keyword.

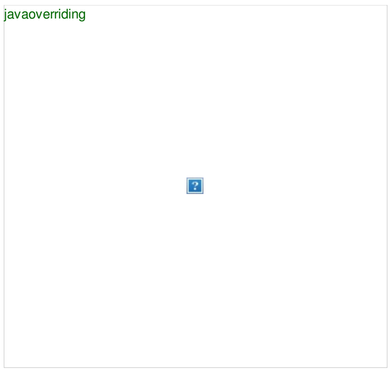
```
// A Simple Java program to demonstrate overriding
// and super keyword in Java

// Base Class
class Parent
{
    void show()
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent
{
    // This method overrides show() of Parent
    void show()
    {
        super.show();
        System.out.println("Child's show()");
    }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        Parent obj = new Child();
        obj.show();
    }
}
```

Output:

```
Parent's show()
Child's show()
```

**Applications**

Overridden methods allow us to call methods of any of the derived classes without even knowing kind of derived class object. For example, consider a employee management software for an organization, let the code has a simple base class Employee, the class has methods like raiseSalary(), transfer(), promote(),.. etc. Different types of employees like Manager, Engineer, ..etc may have their own implementations of the methods present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate functions without even knowing the type of employee. For example, we can easily raise salary of all employees by iterating through list of employees. Every type of employee may have its own logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that function would be called.
We can pass a base class reference to a method and call overridden methods of derived classes without even worrying about types of derived class types.



```
// A Simple Java program to demonstrate application
// of overriding in Java

// Base Class
class Employee
{
    public static int base = 10000;
    int salary()
    {
        return base;
    }
}

// Inherited class
class Manager extends Employee
{
    // This method overrides show() of Parent
    int salary()
    {
        return base + 20000;
    }
}

// Inherited class
class Clerk extends Employee
{
    // This method overrides show() of Parent
    int salary()
    {
        return base + 10000;
    }
}

// Driver class
class Main
{
    // This method can be used to print salary of
    // any type of employee using base class refernce
    static void printSalary(Employee e)
    {
        System.out.println(e.salary());
    }

    public static void main(String[] args)
    {
        Employee obj1 = new Manager();

        // We could also get type of employee using
        // one more overridden method.loke getType()
        System.out.print("Manager's salary : ");
        printSalary(obj1);

        Employee obj2 = new Clerk();
        System.out.print("Clerk's salary : ");
        printSalary(obj2);
    }
}
```

Output:

```
Parent's show()
Child's show()
```

**Overriding vs Overloading**

- Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.

- Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org.

See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## How does default virtual behavior differ in C++ and Java ?

**Default virtual behavior of methods is opposite in C++ and Java:**

In C++, class member methods are non-virtual by default. They can be made virtual by using *virtual* keyword. For example, *Base::show()* is non-virtual in following program and program prints *“Base::show() called”*.

```
#include<iostream>

using namespace std;

class Base {
public:

    // non-virtual by default
    void show() {
        cout<<"Base::show() called";
    }
};

class Derived: public Base {
public:
    void show() {
        cout<<"Derived::show() called";
    }
};

int main()
{
  Derived d;
  Base &b = d;
  b.show();
  getchar();
  return 0;
}
```

Adding *virtual* before definition of *Base::show()* makes program print *“Derived::show() called”*

In Java, methods are virtual by default and can be made non-virtual by using *final* keyword. For example, in the following java program, *show()* is by default virtual and the program prints *“Derived::show() called”*

```
class Base {

    // virtual by default
    public void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

Unlike C++ non-virtual behavior, if we add *final* before definition of show() in *Base* , then the above program fails in compilation.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Accessing Grandparent’s member in Java

**Directly accessing Grandparent’s member in Java:**

Predict the output of following Java program.

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print();  // Trying to access Grandparent's Print()
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output: Compiler Error

There is error in line “super.super.print();”. In Java, a class cannot directly access the grandparent’s members. It is allowed in C++ though. In C++, we can use scope resolution operator (::) to access any ancestor’s member in inheritance hierarchy. ***In Java, we can access grandparent’s members only through the parent class.*** For example, the following program compiles and runs fine.

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        super.Print();
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Output:

```
Grandparent's Print()
Parent's Print()
Child's Print()
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Shadowing of static functions in Java

In Java, if name of a derived class static function is same as base class static function then the derived class static function shadows (or conceals) the base class static function. For example, the following Java code prints *“A.fun()”*

```
// file name: Main.java
class A {
   static void fun() {
      System.out.println("A.fun()");
   }
}

class B extends A {
   static void fun() {
      System.out.println("B.fun()");
   }
}

public class Main {
   public static void main(String args[]) {
      A a = new B();
      a.fun();  // prints A.fun()
   }
}
```

If we make both A.fun() and B.fun() as non-static then the above program would print “B.fun()”.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Can we override private methods in Java?

Let us first consider the following Java program as a simple example of Overriding or Runtime Polymorphism.

```
class Base {
  public void fun() {
     System.out.println("Base fun");
  }
}

class Derived extends Base {
  public void fun() {  // overrides the Base's fun()
     System.out.println("Derived fun");
  }
  public static void main(String[] args) {
      Base obj = new Derived();
      obj.fun();
  }
}
```

The program prints “Derived fun”.

The Base class reference ‘obj’ refers to a derived class object (see expression “Base obj = new Derived()”). When fun() is called on obj, the call is made according to the type of referred object, not according to the reference.

***Is Overiding possible with private methods?***

Predict the output of following program.

```
class Base {
  private void fun() {
     System.out.println("Base fun");
  }
}

class Derived extends Base {
  private void fun() {
     System.out.println("Derived fun");
  }
  public static void main(String[] args) {
      Base obj = new Derived();
      obj.fun();
  }
}
```

We get compiler error “fun() has private access in Base” (See this). So the compiler tries to call base class function, not derived class, means fun() is not overridden.

***An inner class can access private members of its outer class. What if we extend an inner class and create fun() in the inner class?***

An Inner classes can access private members of its outer class, for example in the following program, *fun()* of *Inner* accesses private data member *msg* which is fine by the compiler.

```
/* Java program to demonstrate whether we can override private method
   of outer class inside its inner class */
class Outer {
     private String msg = "GeeksforGeeks";
     private void fun() {
          System.out.println("Outer fun()");
     }

     class Inner extends Outer {
         private void fun()  {
               System.out.println("Accessing Private Member of Outer: " + msg);
         }
     }

     public static void main(String args[])  {

          // In order to create instance of Inner class, we need an Outer
          // class instance. So, first create Outer class instance and then
          // inner class instance.
          Outer o = new Outer();
          Inner  i   = o.new Inner();

          // This will call Inner's fun, the purpose of this call is to
          // show that private members of Outer can be accessed in Inner.
          i.fun();

          // o.fun() calls Outer's fun (No run-time polymorphism).
          o = i;
          o.fun();
     }
}
```

Output:

```
Accessing Private Member of Outer: GeeksforGeeks
Outer fun()
```

In the above program, we created an outer class and an inner class. We extended Inner from Outer and created a method fun() in both Outer and Inner. If we observe our output, then it is clear that the method fun() has not been overriden. It is so because ***private methods are bonded during compile time and it is the type of the reference variable – not the type of object that it refers to – that determines what method to be called.***. As a side note, private methods may be performance-wise better (compared to non-private and non-final methods) due to static binding.

***Comparison With C++***

**1)** In Java, inner Class is allowed to access private data members of outer class. This behavior is same as C++ (See this).

**2)** In Java, methods declared as private can never be overridden, they are in-fact bounded during compile time. This behavior is different from C++. In C++, we can have virtual private methods (See this).

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# What happens when more restrictive access is given to a derived class method in Java?

In Java, it is compiler error to give more restrictive access to a derived class function which overrides a base class function. For example, if there is a function *public void foo()* in base class and if it is overridden in derived class, then access specifier for *foo()* cannot be anything other than public in derived class. If *foo()* is private function in base class, then access specifier for it can be anything in derived class.

Consider the following two programs. Program 1 fails in compilation and program 2 works fine.

**Program 1**

```
// file name: Main.java
class Base {
    public void foo() {}
}

class Derived extends Base {
    private void foo() {} // compiler error
}

public class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}
```

**Program 2**

```
// file name: Main.java
class Base {
    private void foo() {}
}

class Derived extends Base {
    public void foo() {} // works fine
}

public class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Parent and Child classes having same data member in Java

The reference variable of the Parent class is capable to hold its object reference as well as its child object reference.

In Java, methods are virtual by default (See this for details).

What about non-method members. For example, predict the output of following Java program.

```
// A Java program to demonstrate that non-method
// members are accessed according to reference
// type (Unlike methods which are accessed according
// to the referred object)

class Parent
{
    int value = 1000;
    Parent()
    {
        System.out.println("Parent Constructor");
    }
}

class Child extends Parent
{
    int value = 10;
    Child()
    {
        System.out.println("Child Constructor");
    }
}

// Driver class
class Test
{
    public static void main(String[] args)
    {
        Child obj=new Child();
        System.out.println("Reference of Child Type :"
                    + obj.value);

        // Note that doing "Parent par = new Child()"
        // would produce same result
        Parent par = obj;

        // Par holding obj will access the value
        // variable of parent class
        System.out.println("Reference of Parent Type : "
                    + par.value);
    }
}
```

Output:

```
Parent Constructor
Child Constructor
Reference of Child Type : 10
Reference of Parent Type : 1000
```

If a parent reference variable is holding the reference of the child class and we have the “value” variable in both the parent and child class, it will refer to the parent class “value” variable, whether it is holding child class object reference. The reference holding the child class object reference will not be able to access the members (functions or variables) of the child class. It is because compiler uses special run-time polymorphism mechanism only for methods.

It is possible to access child data members using parent pointer with typecasting. Please see last example of this for complete code.

This article is contributed by **Twinkle Tyagi** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## Java and Multiple Inheritance

Multiple Inheritance is a feature of object oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

**Why Java doesn’t support Multiple Inheritance?**

Consider the below Java code. It shows error.

```
// First Parent class
class Parent1
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
```

```
class Parent2
{
    void fun()
    {
        System.out.println("Parent2");
    }
}

// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

Output :

```
Compiler Error
```

From the code, we see that, on calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Parent2's fun() method.

**1. The Diamond Problem:**

```
          GrandParent
           /     \
          /       \
      Parent1      Parent2
          \       /
           \     /
            Test
```

```
// A Grand parent class in diamond
class GrandParent
{
    void fun()
    {
        System.out.println("Grandparent");
    }
}

// First Parent class
class Parent1 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent1");
    }
}

// Second Parent Class
class Parent2 extends GrandParent
{
    void fun()
    {
        System.out.println("Parent2");
    }
}


// Error : Test is inheriting from multiple
// classes
class Test extends Parent1, Parent2
{
   public static void main(String args[])
   {
       Test t = new Test();
       t.fun();
   }
}
```

From the code, we see that: On calling the method fun() using Test object will cause complications such as whether to call Parent1's fun() or Beta's fun() method.

Therefore, in order to avoid such complications Java does not support multiple inheritance of classes.

**2. Simplicity –** Multiple inheritance is not supported by Java using classes , handling the complexity that causes due to multiple inheritance is very complex. It creates problem during various operations like casting, constructor chaining etc and the above all reason is that there are very few scenarios on which we actually need multiple inheritance, so better to omit it for keeping the things simple and straightforward.

**How are above problems handled for Default Methods and Interfaces ?**

Java 8 supports default methods where interfaces can provide default implementation of methods. And a class can implement two or more interfaces. In case both the implemented interfaces contain default methods with same method signature, the implementing class should explicitly specify which default method is to be used or it should override the default method.

```
// A simple Java program to demonstrate multiple
// inheritance through default methods.
interface PI1
{
    // default method
    default void show()
    {
        System.out.println("Default PI1");
    }
}

interface PI2
{
    // Default method
    default void show()
    {
        System.out.println("Default PI2");
    }
}

// Implementation class code
class TestClass implements PI1, PI2
{
    // Overriding default show method
    public void show()
    {
        // use super keyword to call the show
        // method of PI1 interface
        PI1.super.show();

        // use super keyword to call the show
        // method of PI2 interface
        PI2.super.show();
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

```
Default PI1
Default PI2
```

If we remove implementation of default method from “TestClass”, we get compiler error. See this for a sample run.

If there is a diamond through interfaces, then there is no issue if none of the middle interfaces provide implementation of root interface. If they provide implementation, then implementation can be accessed as above using super keyword.

```
// A simple Java program to demonstrate how diamond
// problem is handled in case of default methods

interface GPI
{
    // default method
    default void show()
    {
        System.out.println("Default GPI");
    }
}

interface PI1 extends GPI { }

interface PI2 extends GPI { }

// Implementation class code
class TestClass implements PI1, PI2
{
    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.show();
    }
}
```

Output:

```
Default GPI
```

Also See – http://qa.geeksforgeeks.org/510/why-java-doesnt-support-multiple-inheritance

This article is contributed by **Vishal S**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

---

## Interfaces in Java

Like a class, an interface can have methods and variables, but the methods declared in interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then class must be declared abstract.
- A Java library example is, Comparator Interface. If a class implements this interface, then it can be used to sort a collection.

Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

```
// A simple interface
interface Player
{
    final int id = 10;
    int move();
}
```

To implement an interface we use keyword: implement

```
// Java program to demonstrate working of
// interface.
import java.io.*;

// A simple interface
interface in1
{
    // public, static and final
    final int a = 10;

    // public and abstract
    void display();
}
```

```
// A class that implements interface.
class testClass implements in1
{
    // Implementing the capabilities of
    // interface.
    public void display()
    {
        System.out.println("Geek");
    }

    // Driver Code
    public static void main (String[] args)
    {
        testClass t = new testClass();
        t.display();
        System.out.println(a);
    }
}
```

Output:

```
Geek
10
```

**New features added in interfaces in JDK 8**

1. Prior to JDK 8, interface could not define implementation. We can now add default implementation for interface methods. This default implementation has special use and does not affect the intention behind interfaces.
   Suppose we need to add a new function in an existing interface. Obviously the old code will not work as the classes have not implemented those new functions. So with the help of default implementation, we will give a default body for the newly added functions. Then the old codes will still work.

   ```
   // An example to show that interfaces can
   // have methods from JDK 1.8 onwards
   interface in1
   {
       final int a = 10;
       default void display()
       {
           System.out.println("hello");
       }
   }

   // A class that implements interface.
   class testClass implements in1
   {
       // Driver Code
       public static void main (String[] args)
       {
           testClass t = new testClass();
           t.display();
       }
   }
   ```

   Output :

   ```
   hello
   ```

2. Another feature that was added in JDK 8 is that we can now define static methods in interfaces which can be called independently without an object. Note: these methods are not inherited.

   ```
   // An example to show that interfaces can
   // have methods from JDK 1.8 onwards
   interface in1
   {
       final int a = 10;
       static void display()
       {
           System.out.println("hello");
       }
   }

   // A class that implements interface.
   class testClass implements in1
   {
       // Driver Code
       public static void main (String[] args)
       {
           in1.display();
       }
   }
   ```

   Output :

   ```
   hello
   ```

**Related articles:**

- Access specifier of methods in interfaces
- Access specifiers for classes or interfaces in Java
- Abstract Classes in Java
- Comparator Interface in Java
- Java Interface methods
- Nested Interface in Java

This article is contributed by **Mehak Kumar.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

## Access specifier of methods in interfaces

In Java, all methods in an interface are *public* even if we do not specify *public* with method names. Also, data fields are *public static final* even if we do not mention it with fields names. Therefore, data fields must be initialized.

Consider the following example, *x* is by default *public static final* and *foo()* is *public* even if there are no specifiers.

```
interface Test {
  int x = 10;  // x is public static final and must be initialized here
  void foo();  // foo() is public
}
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Access specifiers for classes or interfaces in Java

In Java, methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

1) private
2) default (when no access specifier is specified)
3) protected
4) public

But, the classes and interfaces themselves can have only two access specifiers when declared outside any other class.
1) public
2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

```
//filename: Main.java
protected class Test {}

public class Main {
  public static void main(String args[]) {

  }
}
```

Note : Nested interfaces and classes can have all access specifiers.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Abstract Classes in Java

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function in C++)
    abstract void draw();
}
```

Following are some important observations about abstract classes in Java.

**1)** Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {

        // Uncommenting the following line will cause compiler error as the
        // line tries to create an instance of abstract class.
        // Base b = new Base();

        // We can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

**2)** Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
    }
}
```

Output:

```
Base Constructor Called
Derived Constructor Called
```

**3)** In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

Output:

```
Base fun() called
```

**4)** Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base {
    final void fun() { System.out.println("Derived fun() called"); }
}

class Derived extends Base {}

class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

**Exercise:**
**1.** Is it possible to create abstract and final class in Java?
**2.** Is it possible to have an abstract method in a final class?
**3.** Is it possible to inherit from multiple abstract classes in Java?

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Comparator Interface in Java with Examples

Comparator interface is used to order the objects of user-defined classes. A comparator object is capable of comparing two objects of two different classes. Following function compare obj1 with obj2

**Syntax:**

```
public int compare(Object obj1, Object obj2):
```

Suppose we have an array/arraylist of our own class type, containing fields like rollno, name, address, DOB etc and we need to sort the array based on Roll no or name?

**Method 1**: One obvious approach is to write our own sort() function using one of the standard algorithms. This solution requires rewriting the whole sorting code for different criterion like Roll No. and Name.

**Method 2:** Using comparator interface- Comparator interface is used to order the objects of user-defined class. This interface is present java.util package and contains 2 methods compare(Object obj1, Object obj2) and equals(Object element). Using comparator, we can sort the elements based on data members. For instance it may be on rollno, name, age or anything else.

Method of Collections class for sorting List elements is used to sort the elements of List by the given comparator.

```
// To sort a given list. ComparatorClass must implement
// Comparator interface.
public void sort(List list, ComparatorClass c)
```

**How does Collections.Sort() work?**
Internally the Sort method does call Compare method of the classes it is sorting. To compare two elements, it asks “Which is greater?” Compare method returns -1, 0 or 1 to say if it is less than, equal, or greater to the other. It uses this result to then determine if they should be swapped for its sort.

**Working Program:**

```
// Java program to demonstrate working of Comparator
// interface
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                           String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                           " " + this.address;
    }
}

class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

class Sortbyname implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll name
    public int compare(Student a, Student b)
    {
        return a.name.compareTo(b.name);
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyname());

        System.out.println("\nSorted by name");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Output:

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc

Sorted by name
131 aaaa nyc
111 bbbb london
121 cccc jaipu
```

By changing the return value in inside compare method you can sort in any order you want. eg.for descending order just change the positions of a and b in above compare method.

**References:**

http://www.dreamincode.net/forums/topic/169079-how-collectionssort-is-doing-its-stuff-here/

http://www.javatpoint.com/Comparator-interface-in-collection-framework

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java
Java

## Java Interface methods

There is a rule that every member of interface is only and only public whether you define or not. So when we define the method of the interface in a class implementing the interface, we have to give it public access as child class can't assign the weaker access to the methods.

```
// A Simple Java program to demonstrate that
// interface methods must be public in
// implementing class
interface A
{
    void fun();
}

class B implements A
{
    // If we change public to anything else,
    // we get compiler error
    public void fun()
    {
        System.out.println("fun()");
    }
}

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

Output:

```
fun()
```

If we change fun() to anything other than public in class B, we get compiler error "attempting to assign weaker access privileges; was public"

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# Nested Interface in Java

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

**Interface in a class**

Interfaces (or classes) can have only public and default access specifiers when declared outside any other class (Refer this for details). This interface declared in a class can either be default, public, private, protected. While implementing the interface, we mention the interface as **c_name.i_name** where **c_name** is the name of the class in which it is nested and **i_name** is the name of the interface itself.
Let us have a look at the following code:-

```
// Java program to demonstrate working of
// interface inside a class.
import java.util.*;
class Test
{
    interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj=t;
        obj.show();
    }
}
```

```
show method of interface
```

The access specifier in above example is default. We can assign public, protected or private also. Below is an example of protected. In this particular example, if we change access specifier to private, we get compiler error because a derived class tries to access it.

```
// Java program to demonstrate protected
// specifier for nested interface.
import java.util.*;
class Test
{
    protected interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj=t;
        obj.show();
    }
}
```

```
show method of interface
```

**Interface in another Interface**

An interface can be declared inside another interface also. We mention the interface as **i_name1.i_name2** where **i_name1** is the name of the interface in which it is nested and **i_name2** is the name of the interface to be implemented.

```
// Java program to demonstrate working of
// interface inside another interface.
import java.util.*;
interface Test
{
   interface Yes
   {
      void show();
   }
}

class Testing implements Test.Yes
{
   public void show()
   {
      System.out.println("show method of interface");
   }
}

class A
{
   public static void main(String[] args)
   {
     Test.Yes obj;
     Testing t = new Testing();
     obj = t;
     obj.show();
   }
}
```

```
show method of interface
```

**Note:** In the above example, access specifier is public even if we have not written public. If we try to change access specifier of interface to anything other than public, we get compiler error. Remember, interface members can only be public..

```
// Java program to demonstrate an interface cannot
// have non-public member interface.
import java.util.*;
interface Test
{
    protected interface Yes
    {
        void show();
    }
}

class Testing implements Test.Yes
{
    public void show()
    {
        System.out.println("show method of interface");
    }
}

class A
{
    public static void main(String[] args)
    {
        Test.Yes obj;
        Testing t = new Testing();
        obj = t;
        obj.show();
    }
}
```

```
illegal combination of modifiers: public and protected
   protected interface Yes
```

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Comparable vs Comparator in Java

Java provides two interfaces to sort objects using data members of the class:

1. Comparable
2. Comparator

**Using Comparable Interface**

A comparable object is capable of comparing itself with another object. The class itself must implements the **java.lang.Comparable** interface to compare its instances.

Consider a Movie class that has members like, rating, name, year. Suppose we wish to sort a list of Movies based on year of release. We can implement the Comparable interface with the Movie class, and we override the method compareTo() of Comparable interface.

```
// A Java program to demonstrate use of Comparable
import java.io.*;
import java.util.*;

// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;
    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }

    // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        Collections.sort(list);

        System.out.println("Movies after sorting : ");
        for (Movie movie: list)
        {
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());
        }
    }
}
```

Output:

```
Movies after sorting :
Star Wars 8.7 1977
Empire Strikes Back 8.8 1980
Return of the Jedi 8.4 1983
Force Awakens 8.3 2015
```

Now, suppose we want sort movies by their rating and names also. When we make a collection element comparable(by having it implement Comparable), we get only one chance to implement the compareTo() method. The solution is using Comparator.

**Using Comparator**

Unlike Comparable, Comparator is external to the element type we are comparing. It's a separate class. We create multiple separate classes (that implement Comparator) to compare by different members.

Collections class has a second sort() method and it takes Comparator. The sort() method invokes the compare() to sort objects.

To compare movies by Rating, we need to do 3 things :

1. Create a class that implements Comparator (and thus the compare() method that does the work previously done by compareTo()).
2. Make an instance of the Comparator class.
3. Call the overloaded sort() method, giving it both the list and the instance of the class that implements Comparator.

```
//A Java program to demonstrate Comparator interface
import java.io.*;
import java.util.*;

// A class 'Movie' that implements Comparable
class Movie implements Comparable<Movie>
{
    private double rating;
    private String name;
    private int year;

    // Used to sort movies by year
    public int compareTo(Movie m)
    {
        return this.year - m.year;
    }

    // Constructor
    public Movie(String nm, double rt, int yr)
    {
        this.name = nm;
        this.rating = rt;
        this.year = yr;
    }

    // Getter methods for accessing private data
    public double getRating() { return rating; }
    public String getName()   {  return name; }
    public int getYear()      {  return year;  }
}

// Class to compare Movies by ratings
class RatingCompare implements Comparator<Movie>
{
    public int compare(Movie m1, Movie m2)
    {
        if (m1.getRating() < m2.getRating()) return -1;
        if (m1.getRating() > m2.getRating()) return 1;
        else return 0;
    }
}

// Class to compare Movies by name
class NameCompare implements Comparator<Movie>
```

```
{
    public int compare(Movie m1, Movie m2)
    {
        return m1.getName().compareTo(m2.getName());
    }
}

// Driver class
class Main
{
    public static void main(String[] args)
    {
        ArrayList<Movie> list = new ArrayList<Movie>();
        list.add(new Movie("Force Awakens", 8.3, 2015));
        list.add(new Movie("Star Wars", 8.7, 1977));
        list.add(new Movie("Empire Strikes Back", 8.8, 1980));
        list.add(new Movie("Return of the Jedi", 8.4, 1983));

        // Sort by rating : (1) Create an object of ratingCompare
        //                  (2) Call Collections.sort
        //                  (3) Print Sorted list
        System.out.println("Sorted by rating");
        RatingCompare ratingCompare = new RatingCompare();
        Collections.sort(list, ratingCompare);
        for (Movie movie: list)
            System.out.println(movie.getRating() + " " +
                               movie.getName() + " " +
                               movie.getYear());


        // Call overloaded sort method with RatingCompare
        // (Same three steps as above)
        System.out.println("\nSorted by name");
        NameCompare nameCompare = new NameCompare();
        Collections.sort(list, nameCompare);
        for (Movie movie: list)
            System.out.println(movie.getName() + " " +
                               movie.getRating() + " " +
                               movie.getYear());

        // Uses Comparable to sort by year
        System.out.println("\nSorted by year");
        Collections.sort(list);
        for (Movie movie: list)
            System.out.println(movie.getYear() + " " +
                               movie.getRating() + " " +
                               movie.getName()+" ");
    }
}
```

Output :

```
Sorted by rating
8.3 Force Awakens 2015
8.4 Return of the Jedi 1983
8.7 Star Wars 1977
8.8 Empire Strikes Back 1980

Sorted by name
Empire Strikes Back 8.8 1980
Force Awakens 8.3 2015
Return of the Jedi 8.4 1983
Star Wars 8.7 1977

Sorted by year
1977 8.7 Star Wars
1980 8.8 Empire Strikes Back
1983 8.4 Return of the Jedi
2015 8.3 Force Awakens
```

- Comparable is meant for objects with natural ordering which means the object itself must know how it is to be ordered. For example Roll Numbers of students. Whereas, Comparator interface sorting is done through a separate class.
- Logically, Comparable interface compares “this” reference with the object specified and Comparator in Java compares two different class objects provided.
- If any class implements Comparable interface in Java then collection of that object either List or Array can be sorted automatically by using Collections.sort() or Arrays.sort() method and objects will be sorted based on there natural order defined by CompareTo method.

**To summarize, if sorting of objects needs to be based on natural order then use Comparable whereas if you sorting needs to be done on attributes of different objects, then use Comparator in Java.**

This article is contributed by **Souradeep Barua.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# Functional Interfaces In Java

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***, ***Comparable*** are some of the examples of functional interfaces.
Before Java 8, we had to create anonymous inner class objects or implement these interfaces.

```
// Java program to demonstrate functional interface

class Test
{
    public static void main(String args[])
    {
        // create anonymous inner class object
        new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                System.out.println("New thread created");
            }
        }).start();
    }
}
```

Output:

```
New thread created
```

Java 8 onwards, we can assign lambda expression to its functional interface object like this:

```
// Java program to demonstrate Implementation of
// functional interface using lambda expressions

class Test
{
  public static void main(String args[])
  {

    // lambda expression to create the object
    new Thread(()->
       {System.out.println("New thread created");}).start();
  }
}
```

```
New thread created
```

**@FunctionalInterface Annotation**

@FunctionalInterface annotation is used to ensure that the functional interface can't have more than one abstract method. In case more than one abstract methods are present, the compiler flags an 'Unexpected @FunctionalInterface annotation' message. However, it is not mandatory to use this annotation.

```
// Java program to demonstrate lamda expressions to implement
// a user defined functional interface.

@FunctionalInterface
interface Square
{
    int calculate(int x);
}

class Test
{
    public static void main(String args[])
    {
        int a = 5;

        // lambda expression to define the calculate method
        Square s = (int x)->x*x;

        // parameter passed and return type must be
        // same as defined in the prototype
        int ans = s.calculate(a);
        System.out.println(ans);
    }
}
```

Output:

```
25
```

**java.util.function Package:**

The java.util.function package in Java 8 contains many builtin functional interfaces like-

- **Predicate:** The Predicate interface has an abstract method test which gives a Boolean value as a result for the specified argument. Its prototype is

  ```
  public Predicate
  {
     public boolean test(T  t);
   }
  ```

- **BinaryOperator:** The BinaryOperator interface has an abstract method apply which takes two argument and returns a result of same type. Its prototype is

  ```
  public interface BinaryOperator
  {
       public T apply(T x, T y);
  }
  ```

- **Function:** The Function interface has an abstract method apply which takes argument of type T and returns a result of type R. Its prototype is

  ```
  public interface Function
  {
     public R apply(T t);
  }
  ```

```
// A simple program to demonstrate the use
// of predicate interface
import java.util.*;
import java.util.function.Predicate;

class Test
{
    public static void main(String args[])
    {

        // create a list of strings
        List<String> names =
            Arrays.asList("Geek","GeeksQuiz","g1","QA","Geek2");

        // declare the predicate type as string and use
        // lambda expression to create object
        Predicate<String> p = (s)->s.startsWith("G");

        // Iterate through the list
        for (String st:names)
        {
            // call the test method
            if (p.test(st))
                System.out.println(st);
        }
    }
```

```
}
```

Output:

```
Geek
GeeksQuiz
Geek2
```

**Important Points/Observations:**

1. A functional interface has only one abstract method but it can have multiple default methods.
2. @FunctionalInterface annotation is used to ensure an interface can't have more than one abstract method. The use of this annotation is optional.
3. The java.util.function package contains many builtin functional interfaces in Java 8.

This article is contributed by **Akash Ojha** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Queue Interface In Java

The java.util.Queue is a subtype of java.util.Collection interface. It is an ordered list of objects with its use limited to inserting elements at the end of list and deleting elements from the start of list i.e. it follows FIFO principle.
Since it is an interface, we need a concrete class during its declaration. There are many ways to initialize a Queue object, most common being-

1. As a Priority Queue
2. As a LinkedList

Please note that both the implementations are not thread safe. PriorityBlockingQueue is one alternative implementation if you need a thread safe implementation.

**Operations on Queue :**

- **Add()-**Adds an element at the tail of queue. More specifically, at the last of linkedlist if it is used, or according to the priority in case of priority queue implementation.
- **peek()-**To view the head of queue without removing it. Returns null if queue is empty.
- **element()-**Similar to peek(). Throws NoSuchElementException if queue is empty.
- **remove()-**Removes and returns the head of the queue. Throws NoSuchElementException when queue is impty.
- **poll()-**Removes and returns the head of the queue. Returns null if queue is empty.

Since it is a subtype of Collections class, it inherits all the methods of it namely **size(), isEmpty(), contains() etc.**

A simple Java program to demonstrate these methods

```
// Java orogram to demonstrate working of Queue
// interface in Java
import java.util.LinkedList;
import java.util.Queue;

public class QueueExample
{
  public static void main(String[] args)
  {
    Queue<Integer> q = new LinkedList<>();

    // Adds elements {0, 1, 2, 3, 4} to queue
    for (int i=0; i<5; i++)
     q.add(i);

    // Display contents of the queue.
    System.out.println("Elements of queue-"+q);

    // To remove the head of queue.
    int removedele = q.remove();
    System.out.println("removed element-" + removedele);

    System.out.println(q);

    // To view the head of queue
    int head = q.peek();
    System.out.println("head of queue-" + head);

    // Rest all methods of collection interface,
    // Like size and contains can be used with this
    // implementation.
    int size = q.size();
    System.out.println("Size of queue-" + size);
  }
}
```

Output:

```
Elements of queue-[0, 1, 2, 3, 4]
removed element-0
[1, 2, 3, 4]
head of queue-1
Size of queue-4
```

Applications of queue data structure can be found here

This article is contributed by **Rishabh Mahrsee** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Multithreading in Java

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :
1. Extending the Thread class
2. Implementing the Runnable Interface

**Thread creation by extending the Thread class**

We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                  Thread.currentThread().getId() +
                  " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
public class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            MultithreadingDemo object = new MultithreadingDemo();
            object.start();
        }
    }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

**Thread creation by implementing the Runnable Interface**

We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

```
// Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable
{
    public void run()
    {
        try
        {
            // Displaying the thread that is running
            System.out.println ("Thread " +
                                Thread.currentThread().getId() +
                                " is running");

        }
        catch (Exception e)
        {
            // Throwing an exception
            System.out.println ("Exception is caught");
        }
    }
}

// Main Class
class Multithread
{
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i=0; i<8; i++)
        {
            Thread object = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

Output :

```
Thread 8 is running
Thread 9 is running
Thread 10 is running
Thread 11 is running
Thread 12 is running
Thread 13 is running
Thread 14 is running
Thread 15 is running
```

**Thread Class vs Runnable Interface**

1. If we extend the Thread class, our class cannot extend any other class because Java doesn't support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.

2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.

This article is contributed by Mehak Narang. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# Synchronized in Java

Multi-threaded programs may often come to a situation where multiple threads try to access the same resources and finally produce erroneous and unforeseen results.

So it needs to be made sure by some synchronization method that only one thread can access the resource at a given point of time.

Java provides a way of creating threads and synchronizing their task by using synchronized blocks. Synchronized blocks in Java are marked with the synchronized keyword. A synchronized block in Java is synchronized on some object. All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time. All other threads attempting to enter the synchronized block are blocked until the thread inside the synchronized block exits the block.

Following is the general form of a synchronized block:

```
// Only one thread can execute at a time.
// sync_object is a reference to an object
// whose lock associates with the monitor.
// The code is said to be synchronized on
// the monitor object
synchronized(sync_object)
{
   // Access shared variables and other
   // shared resources
}
```

This synchronization is implemented in Java with a concept called monitors. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.

Following is an example of multi threading with synchronized.

```
// A Java program to demonstrate working of
// synchronized.
import java.io.*;
import java.util.*;

// A Class used to send a message
class Sender
{
    public void send(String msg)
    {
        System.out.println("Sending\t"  + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread  interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}

// Class for send a message using Threads
class ThreadedSend extends Thread
{
    private String msg;
    private Thread t;
    Sender  sender;

    // Recieves a message object and a string
    // message to be sent
    ThreadedSend(String m,  Sender obj)
    {
        msg = m;
        sender = obj;
    }

    public void run()
    {
        // Only one thread can send a message
        // at a time.
        synchronized(sender)
        {
            // synchronizing the snd object
            sender.send(msg);
        }
    }
}

// Driver class
class SyncDemo
{
    public static void main(String args[])
    {
        Sender snd = new Sender();
        ThreadedSend S1 =
            new ThreadedSend( " Hi " , snd );
        ThreadedSend S2 =
            new ThreadedSend( " Bye " , snd );

        // Start two threads of ThreadedSend type
        S1.start();
        S2.start();

        // wait for threads to end
        try
        {
            S1.join();
            S2.join();
        }
        catch(Exception e)
        {
            System.out.println("Interrupted");
        }
    }
}
```

Output:

```
Sending  Hi

 Hi Sent
Sending  Bye

 Bye Sent
```

The output is same every-time we run the program.

In the above example, we chose to synchronize the Sender object inside the run() method of the ThreadedSend class. Alternately, we could define the **whole send() block as synchronized** and it would produce the same result. Then we don't have to synchronize the Message object inside the run() method in ThreadedSend class.

```
// An alternate implementation to demonstrate
// that we can use synchronized with method also.
class Sender
{
    public synchronized void send(String msg)
    {
        System.out.println("Sending\t" + msg );
        try
        {
            Thread.sleep(1000);
        }
        catch (Exception e)
        {
            System.out.println("Thread interrupted.");
        }
        System.out.println("\n" + msg + "Sent");
    }
}
```

We do not always have to synchronize a whole method. Sometimes it is preferable to **synchronize only part of a method**. Java synchronized blocks inside methods makes this possible.

```
// One more alternate implementation to demonstrate
// that synchronized can be used with only a part of
// method
class Sender
{
    public void send(String msg)
    {
        synchronized(this)
        {
            System.out.println("Sending\t" + msg );
            try
            {
                Thread.sleep(1000);
            }
            catch (Exception e)
            {
                System.out.println("Thread interrupted.");
            }
            System.out.println("\n" + msg + "Sent");
        }
    }
}
```

This article is contributed by **Souradeep Barua**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# Inter-thread Communication in Java

Prerequisite : Multithreading in Java, Synchronized in Java

**What is Polling and what are problems with it?**
The process of testing a condition repeatedly till it becomes true is known as polling.

Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This waste many CPU cycles and makes the implementation inefficient. For example, in a classic queuing problem where one thread is producing data and other is consuming it.

**How Java multi threading tackles this problem?**
To avoid polling, Java uses three methods, namely, **wait(), notify() and notifyAll().**
All these methods belong to object class as final so that all classes have them. They must be used within a synchronized block only.

- **wait()-**It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().

- **notify()-**It wakes up one single thread that called wait() on the same object. It should be noted that calling notify() does not actually give up a lock on a resource.
- **notifyAll()-**It wakes up all the threads that called wait() on the same object.

**A simple Java program to demonstrate the three methods-**
Please note that this program might only run in offline IDEs as it contains taking input at several points.

```
// Java program to demonstrate inter-thread communication
// (wait(), join() and notify()) in Java
import java.util.Scanner;
public class threadexample
{
    public static void main(String[] args)
                        throws InterruptedException
    {
        final PC pc = new PC();

        // Create a thread object that calls pc.produce()
        Thread t1 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                try
                {
                    pc.produce();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });

        // Create another thread object that calls
        // pc.consume()
        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                try
                {
                    pc.consume();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // t1 finishes before t2
        t1.join();
        t2.join();
    }

    // PC (Produce Consumer) class with produce() and
    // consume() methods.
    public static class PC
    {
        // Prints a string and waits for consume()
        public void produce()throws InterruptedException
        {
            // synchronized block ensures only one thread
            // running at a time.
            synchronized(this)
            {
                System.out.println("producer thread running");

                // releases the lock on shared resource
                wait();

                // and waits till some other method invokes notify().
                System.out.println("Resumed");
            }
        }

        // Sleeps for some time and waits for a key press. After key
        // is pressed, it notifies produce().
        public void consume()throws InterruptedException
        {
            // this makes the produce thread to run first.
            Thread.sleep(1000);
            Scanner s = new Scanner(System.in);

            // synchronized block ensures only one thread
            // running at a time.
            synchronized(this)
            {
                System.out.println("Waiting for return key.");
                s.nextLine();
                System.out.println("Return key pressed");

                // notifies the produce thread that it
                // can wake up.
                notify();

                // Sleep
                Thread.sleep(2000);
            }
        }
    }
}
```

Output:

```
producer thread running
```

```
Waiting for return key.
Return key pressed
Resumed
```

As monstrous as it seems, it really is a piece of cake if you go through it twice.

1. In the main class a new PC object is created.
2. It runs produce and consume methods of PC object using two different threads namely t1 and t2 and wait for these threads to finish.

Lets understand how our produce and consume method works.

- First of all, use of synchronized block ensures that only one thread at a time runs. Also since there is a sleep method just at the beginning of consume loop, the produce thread gets a kickstart.
- When the wait is called in produce method, it does two things. Firstly it releases the lock it holds on PC object. Secondly it makes the produce thread to go on a waiting state until all other threads have terminated, that is it can again acquire a lock on PC object and some other method wakes it up by invoking notify or notifyAll on the same object.
- Therefore we see that as soon as wait is called, the control transfers to consume thread and it prints -"Waiting for return key".
- After we press the return key, consume method invokes notify(). It also does 2 things- Firstly, unlike wait(), it does not releases the lock on shared resource therefore for getting the desired result, it is advised to use notify only at the end of your method. Secondly, it notifies the waiting threads that now they can wake up but only after the current method terminates.
- As you might have observed that even after notifying, the control does not immediately passes over to the produce thread. The reason for it being that we have called Thread.sleep() after notify(). As we already know that the consume thread is holding a lock on PC object, another thread cannot access it until it has released the lock. Hence only after the consume thread finishes its sleep time and thereafter terminates by itself, the produce thread cannot take back the control.
- After a 2 second pause, the program terminates to its completion.

If you are still confused as to why we have used notify in consume thread, try removing it and running your program again. As you must have noticed now that the program never terminates.
The reason for this is straightforward-When you called wait on produce thread, it went on waiting and never terminated. Since a program runs till all its threads have terminated, it runs on and on.
There is a second way round this problem. You can use a second variant of wait().

```
void wait(long timeout)
```

This would make the calling thread sleep only for a time specified.

**References :**
Java 2 Complete Reference

This article is contributed by **Rishabh Mahrsee** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Multithreading

# Producer-Consumer solution using threads in Java

In computing, the producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, which share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again.
- At the same time, the consumer is consuming the data (i.e. removing it from the buffer), one piece at a time.

**Problem**
To make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

**Solution**
The producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.
An inadequate solution could result in a deadlock where both processes are waiting to be awakened.

**Recommended Reading-** Multithreading in JAVA, Synchronized in JAVA , Inter-thread Communication

**Implementation of Producer Consumer Class**

- A **LinkedList list** – to store list of jobs in queue.
- **A Variable Capacity** – to check for if the list is full or not
- A mechanism to control the insertion and extraction from this list so that we do not insert into list if it is full or remove from it if it is empty.

*Note: It is recommended to test the below program on a offline IDE as infinite loops and sleep method may lead to it time out on any online IDE*

```
// Java program to implement solution of producer
// consumer problem.
import java.util.LinkedList;

public class Threadexample
{
    public static void main(String[] args)
                        throws InterruptedException
    {
        // Object of a class that has both produce()
        // and consume() methods
        final PC pc = new PC();

        // Create producer thread
        Thread t1 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                try
                {
                    pc.produce();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });

        // Create consumer thread
        Thread t2 = new Thread(new Runnable()
        {
            @Override
            public void run()
            {
                try
                {
                    pc.consume();
                }
                catch(InterruptedException e)
                {
                    e.printStackTrace();
                }
            }
        });

        // Start both threads
        t1.start();
        t2.start();

        // t1 finishes before t2
        t1.join();
        t2.join();
    }

    // This class has a list, producer (adds items to list
    // and consumber (removes items).
    public static class PC
    {
        // Create a list shared by producer and consumer
        // Size of list is 2.
        LinkedList<Integer> list = new LinkedList<>();
        int capacity = 2;

        // Function called by producer thread
        public void produce() throws InterruptedException
        {
            int value = 0;
            while (true)
            {
                synchronized (this)
                {
                    // producer thread waits while list
                    // is full
                    while (list.size()==capacity)
                        wait();

                    System.out.println("Producer produced-"
                                                  + value);

                    // to insert the jobs in the list
                    list.add(value++);

                    // notifies the consumer thread that
                    // now it can start consuming
                    notify();

                    // makes the working of program easier
                    // to  understand
                    Thread.sleep(1000);
                }
            }
        }

        // Function called by consumer thread
        public void consume() throws InterruptedException
        {
            while (true)
            {
                synchronized (this)
                {
                    // consumer thread waits while list
                    // is empty
                    while (list.size()==0)
                        wait();

                    //to retrive the ifrst job in the list
                    int val = list.removeFirst();

                    System.out.println("Consumer consumed-"
                                                    + val);

                    // Wake up producer thread
                    notify();

                    // and sleep
                    Thread.sleep(1000);
                }
            }
        }
    }
}
```

Output:

```
Producer produced-0
Producer produced-1
Consumer consumed-0
Consumer consumed-1
Producer produced-2
```

**Important Points**

- In **PC class** (A class that has both produce and consume methods), a linked list of jobs and a capacity of the list is added to check that producer does not produce if the list is full.
- In **Producer class**, the value is initialized as 0.
  - Also, we have an infinite outer loop to insert values in the list. Inside this loop, we have a synchronized block so that only a producer or a consumer thread runs at a time.
  - An inner loop is there before adding the jobs to list that checks if the job list is full, the producer thread gives up the intrinsic lock on PC and goes on the waiting state.
  - If the list is empty, the control passes to below the loop and it adds a value in the list.

- In the **Consumer class**, we again have an infinite loop to extract a value from the list.
  - Inside, we also have an inner loop which checks if the list is empty.
  - If it is empty then we make the consumer thread give up the lock on PC and passes the control to producer thread for producing more jobs.
  - If the list is not empty, we go round the loop and removes an item from the list.

- In both the methods, we use notify at the end of all statements. The reason is simple, once you have something in list, you can have the consumer thread consume it, or if you have consumed something, you can have the producer produce something.
- sleep() at the end of both methods just make the output of program run in step wise manner and not display everything all at once so that you can see what actually is happening in the program.

**Exercise** :

- Readers are advised to use if condition in place of inner loop for checking boundary conditions.
- Try to make your program produce one item and immediately after that the consumer consumes it before any other item is produced by the consumer.

Reference – https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

## CountDownLatch in Java

CountDownLatch is used to make sure that a task waits for other threads before it starts. To understand its application, let us consider a server where the main task can only start when all the required services have started.

**Working of CountDownLatch:**
When we create an object of CountDownLatch, we specify the number if threads it should wait for, all such thread are required to do count down by calling CountDownLatch.countDown() once they are completed or ready to the job. As soon as count reaches zero, the waiting task starts running.

**Example of CountDownLatch in JAVA:**

```
/* Java Program to demonstrate how to use CountDownLatch,
   Its used when a thread needs to wait for other threads
   before starting its work. */
import java.util.concurrent.CountDownLatch;

public class CountDownLatchDemo
{
    public static void main(String args[]) throws InterruptedException
    {
        // Let us create task that is going to wait for four
        // threads before it starts
        CountDownLatch latch = new CountDownLatch(4);

        // Let us create four worker threads and start them.
        Worker first = new Worker(1000, latch, "WORKER-1");
        Worker second = new Worker(2000, latch, "WORKER-2");
        Worker third = new Worker(3000, latch, "WORKER-3");
        Worker fourth = new Worker(4000, latch, "WORKER-4");
        first.start();
        second.start();
        third.start();
        fourth.start();

        // The main task waits for four threads
        latch.await();

        // Main thread has started
        System.out.println(Thread.currentThread().getName() +
                           " has finished");
    }
}

// A class to represent threads for which the main thread
// waits.
class Worker extends Thread
{
    private int delay;
    private CountDownLatch latch;

    public Worker(int delay, CountDownLatch latch,
                                    String name)
    {
        super(name);
        this.delay = delay;
        this.latch = latch;
    }

    @Override
    public void run()
    {
        try
        {
            Thread.sleep(delay);
            latch.countDown();
            System.out.println(Thread.currentThread().getName()
                            + " finished");
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
    }
}
```

**Output:**

```
WORKER-1 finished
WORKER-2 finished
WORKER-3 finished
WORKER-4 finished
main has finished
```

**Facts about CountDownLatch:**

1. Creating an object of CountDownLatch by passing an int to its constructor (the count), is actually number of invited parties (threads) for an event.
2. The thread, which is dependent on other threads to start processing, waits on until every other thread has called count down. All threads, which are waiting on await() proceed together once count down reaches to zero.
3. countDown() method decrements the count and await() method blocks until count == 0

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Multithreading

# What does start() function do in multithreading in Java?

We have discussed that Java threads are typically created using one of the two methods : (1) Extending thread class. (2) Implementing Runnable

In both the approaches, we override the run() function, but we start a thread by calling the start() function. So why don't we directly call the oveerridden run() function? Why always the start function is called to execute a thread?

**What happens when a function is called?**
When a function is called the following operations take place:

1. The arguments are evaluated.
2. A new stack frame is pushed into the call stack.
3. Parameters are initialized.
4. Method body is executed.
5. Value is retured and current stack frame is popped from the call stack.

**The purpose of start() is to create a separate call stack for the thread. A separate call stack is created by it, and then run() is called by JVM.**

Let us see what happens if we don't call start() and rather call run() directly. We have modified the first program discussed here.

```
// Java code to see that all threads are
// pushed on same stack if we use run()
// instead of start().
class ThreadTest extends Thread
{
  public void run()
  {
    try
    {
      // Displaying the thread that is running
      System.out.println ("Thread " +
           Thread.currentThread().getId() +
           " is running");

    }
    catch (Exception e)
    {
      // Throwing an exception
      System.out.println ("Exception is caught");
    }
  }
}

// Main Class
public class Main
{
  public static void main(String[] args)
  {
    int n = 8;
    for (int i=0; i<n; i++)
    {
      ThreadTest object = new ThreadTest();

      // start() is replaced with run() for
      // seeing the purpose of start
      object.run();
    }
  }
}
```

Output:

```
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
Thread 1 is running
```

We can see from above output that we get same ids for all threads because we have directly called run(). The program that calls start() prints different ids (see this)

**References:**

- http://www.javatpoint.com/what-if-we-call-run()-method-directly
- http://www.leepoint.net/JavaBasics/methods/methods-25-calls.html

This article is contributed by **kp93**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Multithreading

# Java Thread Priority in Multithreading

In a Multi threading environment, thread scheduler assigns processor to a thread based on priority of thread. Whenever we create a thread in Java, it always has some priority assigned to it. Priority can either be given by JVM while creating the thread or it can be given by programmer explicitly.
Accepted value of priority for a thread is in range of 1 to 10. There are 3 static variables defined in Thread class for priority.

**public static int MIN_PRIORITY:** This is minimum priority that a thread can have. Value for this is 1.
**public static int NORM_PRIORITY:** This is default priority of a thread if do not explicitly define it. Value for this is 5.
**public static int MAX_PRIORITY:** This is maximum priority of a thread. Value for this is 10.

**Get and Set Thread Priority:**

1. **public final int getPriority():** java.lang.Thread.getPriority() method returns priority of given thread.
2. **public final void setPriority(int newPriority):** java.lang.Thread.setPriority() method changes the priority of thread to the value newPriority. This method throws IllegalArgumentException if value of parameter newPriority goes beyond minimum(1) and maximum(10) limit.

**Examples of getPriority() and set**

```
// Java program to demonstrate getPriority() and setPriority()
import java.lang.*;

class ThreadDemo extends Thread
{
    public void run()
    {
        System.out.println("Inside run method");
    }

    public static void main(String[]args)
    {
        ThreadDemo t1 = new ThreadDemo();
        ThreadDemo t2 = new ThreadDemo();
        ThreadDemo t3 = new ThreadDemo();

        System.out.println("t1 thread priority : " +
                          t1.getPriority()); // Default 5
        System.out.println("t2 thread priority : " +
                          t2.getPriority()); // Default 5
        System.out.println("t3 thread priority : " +
                          t3.getPriority()); // Default 5

        t1.setPriority(2);
        t2.setPriority(5);
        t3.setPriority(8);

        // t3.setPriority(21); will throw IllegalArgumentException
        System.out.println("t1 thread priority : " +
                          t1.getPriority());  //2
        System.out.println("t2 thread priority : " +
                          t2.getPriority()); //5
        System.out.println("t3 thread priority : " +
                          t3.getPriority());//8

        // Main thread
        System.out.print(Thread.currentThread().getName());
        System.out.println("Main thread priority : "
                     + Thread.currentThread().getPriority());

        // Main thread priority is set to 10
        Thread.currentThread().setPriority(10);
        System.out.println("Main thread priority : "
                     + Thread.currentThread().getPriority());
    }
}
```

```
Output:
t1 thread priority : 5
t2 thread priority : 5
t3 thread priority : 5
t1 thread priority : 2
t2 thread priority : 5
t3 thread priority : 8
Main thread priority : 5
Main thread priority : 10
```

**Note:**

- Thread with highest priority will get execution chance prior to other threads. Suppose there are 3 threads t1, t2 and t3 with priorities 4, 6 and 1. So, thread t2 will execute first based on maximum priority 6 after that t1 will execute and then t3.
- Default priority for main thread is always 5, it can be changed later. Default priority for all other threads depends on the priority of parent thread.
  **Example:**

```
// Java program to demonstrat ethat a child thread
// gets same priority as parent
import java.lang.*;

class ThreadDemo extends Thread
{
    public void run()
    {
        System.out.println("Inside run method");
    }

    public static void main(String[]args)
    {
        // main thread priority is 6 now
        Thread.currentThread().setPriority(6);

        System.out.println("main thread priority : " +
                   Thread.currentThread().getPriority());

        ThreadDemo t1 = new ThreadDemo();

        // t1 thread is child of main thread
        // so t1 thread will also have priority 6.
        System.out.println("t1 thread priority : "
                               + t1.getPriority());
    }
}
```

```
Output:
Main thread priority : 6
t1 thread priority : 6
```

- If two threads have same priority then we can’t expect which thread will execute first. It depends on thread scheduler’s algorithm(Round-Robin, First Come First Serve, etc)
- If we are using thread priority for thread scheduling then we should always keep in mind that underlying platform should provide support for scheduling based on thread priority.

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Garbage Collection in Java

In C/C++, it is programmer's responsibility to delete a dynamically allocated object if it is no longer in use.

In Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects, but the garbage collector is **not guaranteed to run at any specific time**, it can be at any time once an object is eligible for garbage collection.

Following are some important methods/topics related to garbage collection.

**The finalize () method:**

Called by the garbage collector on an object when garbage collector determines that there are no more references to the object
**Note :**

1. The finalize method is never invoked more than once by a Java virtual machine for any given object.
2. Our program must not rely on the finalize method because we never know if finalize will be executed or not.

**Ways to make an object eligible for garbage collection**

Once the object is no longer used by the program, we can change the reference variable to a null, thus making the object which was referred by this variable eligible for garbage collection.
*Please note that the object can not become a candidate for garbage collection until **all** references to it are discarded.*

```
class Test
{
    public static void main(String[] args)
    {
        Test o1 = new Test();

        /* o1 being used for some purpose in program */

        /* When there is no more use of o1, make the object
           referred by o1 eligible for garbage collection */
        o1 = null;

        /* Rest of the program */
     }
}
```

**gc() – request to JVM**

We can request to run the garbage collector using java.lang.System.gc() but it does not force garbage collection, the JVM will run garbage collection only when it wants to run it.
We may use system.gc() or runtime.gc()

```
import java.lang.*;
public class Test
{
    public static void main(String[] args)
    {
        int g1[] = { 0, 1, 2, 3, 4, 5 };
        System.out.println(g1[1] + " ");

        // Requesting Garbage Collector
        System.gc();
        System.out.println("Hey I just requested "+
                          "for Garbage Collection");
    }
}
```

Output:

```
Hey I just requested for Garbage Collection
```

**What happens when group of object only refer to each other?**

It is possible that a set of unused objects only refer to each other. This is also referred as Island of Isolation. For example, object o1 refers to object o2. Object o2 refers to o1. None of them is referenced by any other object. In this case both the objects o1 and o2 are eligible for garbage collection.

```
public class Test
{
    Test geek;
    public static void main(String [] args)
    {
        Test o1 = new Test();
        Test o2 = new Test();
        o1.geek = o2;
        o2.geek = o1;

        o1 = null;
        o2 = null;
       // both become eligible for garbage collection
    }
}
```

This article is contributed by **Chirag Agarwal** .Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# Mark-and-Sweep: Garbage Collection Algorithm

**Background**

All the objects which are created dynamically (using new in C++ and Java) are allocated memory in the heap. If we go on creating objects we might get Out Of Memory error, since it is not possible to allocate heap memory to objects. So we need to clear heap memory by releasing memory for all those objects which are no longer referenced by the program (or the unreachable objects) so that the space is made available for subsequent new objects. This memory can be released by the programmer itself but it seems to be an overhead for the programmer, here garbage collection comes to our rescue, and it automatically releases the heap memory for all the unreferenced objects.

There are many garbage collection algorithms which run in the background. One of them is mark and sweep.

**Mark and Sweep Algorithm**

Any garbage collection algorithm must perform 2 basic operations. One, it should be able to detect all the unreachable objects and secondly, it must reclaim the heap space used by the garbage objects and make the space available again to the program.

The above operations are performed by Mark and Sweep Algorithm in two phases:

1) Mark phase

2) Sweep phase

**Mark Phase**

When an object is created, its mark bit is set to 0(false). In the Mark phase, we set the marked bit for all the reachable objects (or the objects which a user can refer to) to 1(true). Now to perform this operation we simply need to do a graph traversal, a depth first search approach would work for us. Here we can consider every object as a node and then all the nodes (objects) that are reachable from this node (object) are visited and it goes on till we have visited all the reachable nodes.

- Root is a variable that refer to an object and is directly accessible by local variable. We will assume that we have one root only.
- We can access the mark bit for an object by: markedBit(obj).

Algorithm -Mark phase:

```
Mark(root)
  If markedBit(root) = false then
    markedBit(root) = true
    For each v referenced by root
       Mark(v)
```

*Note: If we have more than one root, then we simply have to call Mark() for all the root variables.*

**Sweep Phase**

As the name suggests it “sweeps” the unreachable objects i.e. it clears the heap memory for all the unreachable objects. All those objects whose marked value is set to false are cleared from the heap memory, for all other objects (reachable objects) the marked bit is set to false.

Now the mark value for all the reachable objects is set to false, since we will run the algorithm (if required) and again we will go through the mark phase to mark all the reachable objects.

Algorithm – Sweep Phase

```
Sweep()
For each object p in heap
  If markedBit(p) = true then
    markedBit(p) = false
  else
    heap.release(p)
```

The mark-and-sweep algorithm is called a tracing garbage collector because is traces out the entire collection of objects that are directly or indirectly accessible by the program.

Example:

a) All the objects have their marked bits set to false.

M&S_Fig1

M&S_Fig2

b) Reachable objects are marked true

c) Non reachable objects are cleared from the heap.

M&S_Fig3

**Advantages of Mark and Sweep Algorithm**

- It handles the case with cyclic references, even in case of a cycle, this algorithm never ends up in an infinite loop.
- There are no additional overheads incurred during the execution of the algorithm.

**Disadvantages of Mark and Sweep Algorithm**

- The main disadvantage of the mark-and-sweep approach is the fact that that normal program execution is suspended while the garbage collection algorithm runs.
- Other disadvantage is that, after the Mark and Sweep Algorithm is run several times on a program, reachable objects end up being separated by many, small unused memory regions. Look at the below figure for better understanding.

HeapMemory

Figure:

Here white blocks denote the free memory, while the grey blocks denote the memory taken by all the reachable objects.
Now the free segments (which are denoted by white color) are of varying size let's say the 5 free segments are of size 1, 1, 2, 3, 5 (size in units).
Now we need to create an object which takes 10 units of memory, now assuming that memory can be allocated only in contiguous form of blocks, the creation of object isn't possible although we have an available memory space of 12 units and it will cause OutOfMemory error. This problem is termed as "Fragmentation". We have memory available in "fragments" but we are unable to utilize that memory space.
We can reduce the fragmentation by compaction; we shuffle the memory content to place all the free memory blocks together to form one large block. Now consider the above example, after compaction we have a continuous block of free memory of size 12 units so now we can allocate memory to an object of size 10 units.

**References:**

- Data Structures and Algorithms with Object-Oriented Design Patterns in Java
- https://en.wikipedia.org/wiki/Tracing_garbage_collection#Na.C3.AFve_mark-and-sweep
- https://blogs.msdn.microsoft.com/abhinaba/2009/01/30/back-to-basics-mark-and-sweep-garbage-collection/

# Island of Isolation in Java

In java, object destruction is taken care by the Garbage Collector module and the objects which do not have any references to them are eligible for garbage collection. Garbage Collector is capable to identify this type of objects.

**Island of Isolation:**

- Object 1 references Object 2 and Object 2 references Object 1. Neither Object 1 nor Object 2 is referenced by any other object. That's an island of isolation.
- Basically, an island of isolation is a group of objects that reference each other but they are not referenced by any active object in the application. Strictly speaking, even a single unreferenced object is an island of isolation too.

**Example:**

```
public class Test
{
    Test i;
    public static void main(String[] args)
    {
        Test t1 = new Test();
        Test t2 = new Test();

        // Object of t1 gets a copy of t2
        t1.i = t2;

        // Object of t2 gets a copy of t1
        t2.i = t1;

        // Till now no object eligible
        // for garbage collection
        t1 = null;

        //now two objects are eligible for
        // garbage collection
        t2 = null;

        // calling garbage collector
        System.gc();

    }

    @Override
    protected void finalize() throws Throwable
    {
        System.out.println("Finalize method called");
    }
}
```

Output:

```
Finalize method called
Finalize method called
```

**Explanation :**
Before destructing an object, Garbage Collector calls finalize method at most one time on that object.
The reason finalize method called two times in above example because two objects are eligible for garbage collection.This is because we don't have any external references to t1 and t2 objects after executing t2=null.
All we have is only internal references(which is in instance variable i of class Test) to them of each other. There is no way we can call instance variable of both objects. So, none of the objects can be called again.

**Till t2.i = t1 :** Both the objects have external references t1 and t2.

Untitled_1

**t1 = null :** Both the objects can be reached via t2.i and t2 respectively.

Untitled_1

**t2 = null:** No way to reach any of the objects.

Untitled_1

Now, both the objects are eligible for garbage collection as **there is no way we can call them**. This is popularly known as **Island of Isolation.**

**Reference:**

- http://stackoverflow.com/questions/792831/island-of-isolation-of-garbage-collection

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Reflection in Java

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

- The required classes for reflection are provided under java.lang.reflect package.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

reflection

Reflection can be used to get information about –

1. **Class** The getClass() method is used to get the name of the class to which an object belongs.
2. **Constructors** The getConstructors() method is used to get the public constructors of the class to which an object belongs.
3. **Methods** The getMethods() method is used to get the public methods of the class to which an objects belongs.

```
// A simple Java program to demonstrate the use of reflection
import java.lang.reflect.Method;
import java.lang.reflect.Field;
import java.lang.reflect.Constructor;

// class whose object is to be created
class Test
{
    // creating a private field
    private String s;

    // creating a public constructor
    public Test()  {  s = "GeeksforGeeks"; }

    // Creating a public method with no arguments
    public void method1()  {
        System.out.println("The string is " + s);
    }

    // Creating a public method with int as argument
    public void method2(int n)  {
        System.out.println("The number is " + n);
    }

    // creating a private method
    private void method3() {
        System.out.println("Private method invoked");
    }
}

class Demo
{
    public static void main(String args[]) throws Exception
    {
        // Creating object whose property is to be checked
        Test obj = new Test();

        // Creating class object from the object using
        // getclass method
        Class cls = obj.getClass();
        System.out.println("The name of class is " +
                            cls.getName());

        // Getting the constructor of the class through the
        // object of the class
        Constructor constructor = cls.getConstructor();
        System.out.println("The name of constructor is " +
                            constructor.getName());

        System.out.println("The public methods of class are : ");

        // Getting methods of the class through the object
        // of the class by using getMethods
        Method[] methods = cls.getMethods();

        // Printing method names
        for (Method method:methods)
            System.out.println(method.getName());

        // creates object of desired method by providing the
        // method name and parameter class as arguments to
        // the getDeclaredMethod
        Method methodcall1 = cls.getDeclaredMethod("method2",
                                                 int.class);

        // invokes the method at runtime
        methodcall1.invoke(obj, 19);

        // creates object of the desired field by providing
        // the name of field as argument to the
        // getDeclaredField method
        Field field = cls.getDeclaredField("s");

        // allows the object to access the field irrespective
        // of the access specifier used with the field
        field.setAccessible(true);

        // takes object and the new value to be assigned
        // to the field as arguments
        field.set(obj, "JAVA");

        // Creates object of desired method by providing the
        // method name as argument to the getDeclaredMethod
        Method methodcall2 = cls.getDeclaredMethod("method1");

        // invokes the method at runtime
        methodcall2.invoke(obj);

        // Creates object of the desired method by providing
        // the name of method as argument to the
        // getDeclaredMethod method
        Method methodcall3 = cls.getDeclaredMethod("method3");

        // allows the object to access the method irrespective
        // of the access specifier used with the method
        methodcall3.setAccessible(true);

        // invokes the method at runtime
        methodcall3.invoke(obj);
    }
}
```

Output :

```
The name of class is Test
The name of constructor is Test
The public methods of class are :
method2
method1
wait
wait
wait
equals
toString
hashCode
getClass
notify
notifyAll
The number is 19
The string is JAVA
Private method invoked
```

**Important observations :**

1. We can invoke an method through reflection if we know its name and parameter types. We use below two methods for this purpose
   **getDeclaredMethod() :** To create an object of method to be invoked. The syntax for this method is

   ```
   Class.getDeclaredMethod(name, parametertype)
   name- the name of method whose object is to be created
   parametertype - parameter is an array of Class objects
   ```

   **invoke() :** To invoke a method of the class at runtime we use following method–

   ```
   Method.invoke(Object, parameter)
   If the method of the class doesn't accepts any
   parameter then null is passed as argument.
   ```

2. Through reflection we can **access the private variables and methods** of a class with the help of its class object and invoke the method by using the object as discussed above. We use below two methods for this purpose.
   **Class.getDeclaredField(FieldName) :** Used to get the private field. Returns an object of type Field for specified field name.
   **Field.setAccessible(true) :** Allows to access the field irrespective of the access modifier used with the field.

**Advantages of Using Reflection:**

- **Extensibility Features:** An application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
- **Debugging and testing tools**: Debuggers use the property of reflection to examine private members on classes.

**Drawbacks:**

- **Performance Overhead:** Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- **Exposure of Internals:** Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

**Reference:**
https://docs.oracle.com/javase/tutorial/reflect/index.html

This article is contributed by **Akash Ojha**.If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java
Java

## new operator vs newInstance() method in Java

In general, new operator is used to create objects, but if we want to decide type of object to be created at runtime, there is no way we can use new operator. In this case, we have to use newInstance() method. Consider an example:

```
// Java program to demonstrate working of newInstance()

// Sample classes
class A {  int a; }
class B {  int b; }

public class Test
{
    // This method creates an instance of class whose name is
    // passed as a string 'c'.
    public static void fun(String c)  throws InstantiationException,
        IllegalAccessException, ClassNotFoundException
    {
        // Create an object of type 'c'
        Object obj = Class.forName(c).newInstance();

        // This is to print type of object created
        System.out.println("Object created for class:"
                        + obj.getClass().getName());
    }
```

```
    // Driver code that calls main()
    public static void main(String[] args) throws InstantiationException,
    IllegalAccessException, ClassNotFoundException
    {
        fun("A");
    }
}
```

Output:

```
Object created for class:A
```

**Class.forName()** method return class **Class** object on which we are calling **newInstance()** method which will return the object of that class which we are passing as command line argument.
If the passed class doesn't exist then **ClassNotFoundException** will occur.
**InstantionException** will occur if the passed class doesn't contain default constructor as **newInstance()** method internally calls the default constructor of that particular class.
**IllegalAccessException** will occur if we don't have access to the definition of specified class definition.

**Related Article:** Reflection in Java

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

---

## instanceof operator vs isInstance() method in Java

**instanceof** operator and **isInstance()** method both are used for checking the class of the object. But main difference comes when we want to check the class of object dynamically. In this case **isInstance()** method will work. There is no way we can do this by **instanceof** operator.

**instanceof** operator and **isInstance()** method both return a boolean value. Consider an example:

```
// Java program to demonstrate working of
// instanceof operator
public class Test
{
    public static void main(String[] args)
    {
        Integer i = new Integer(5);

        // prints true as i is instance of class
        // Integer
        System.out.println(i instanceof Integer);
    }
}
```

Output:

```
true
```

Now if we want to check the class of the object at run time, then we must use **isInstance()** method.

```
// Java program to demonstrate working of isInstance()
// method
public class Test
{
    // This method tells us whether the object is an
    // instance of class whose name is passed as a
    // string 'c'.
    public static boolean fun(Object obj, String c)
                   throws ClassNotFoundException
    {
        return Class.forName(c).isInstance(obj);
    }

    // Driver code that calls fun()
    public static void main(String[] args)
                   throws ClassNotFoundException
    {
        Integer i = new Integer(5);

        // print true as i is instance of class
        // Integer
        boolean b = fun(i, "java.lang.Integer");

        // print false as i is not instance of class
        // String
        boolean b1 = fun(i, "java.lang.String");

        /* print true as i is also instance of class
           Number as Integer class extends Number
           class*/
        boolean b2 = fun(i, "java.lang.Number");

        System.out.println(b);
        System.out.println(b1);
        System.out.println(b2);
    }
}
```

Output:

```
true
false
true
```

**NOTE:** instanceof operator throws compile time error(Incompatible conditional operand types) if we check object with other classes which it doesn't instantiate.

```
public class Test
{
    public static void main(String[] args)
    {
        Integer i = new Integer(5);

        // Below line causes compile time error:-
        // Incompatible conditional operand types
        // Integer and String
        System.out.println(i instanceof String);
    }
}
```

Output :

```
13: error: incompatible types: Integer cannot be converted to String
        System.out.println(i instanceof String);
                       ^
```

**Related Articles:**
new operator vs newInstance() method in Java
Reflections in Java

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Redirecting System.out.println() output to a file in Java

*System.out.println()* is used mostly to print messages to the console. However very few of us are actually aware of its working mechanism.

- *System* is a class defined in the *java.lang* package.
- out is an instance of *PrintStream* , which is a public and static member of the class *System*.
- As all instances of *PrintStream* class have a public method *println()*, hence we can invoke the same on out as well. We can assume *System.out* represents the standard Output Stream .

One interesting fact related to the above topic is, **we can use *System.out.println()* to print messages to other sources too (and not just console)** . However before doing so , we must reassign the standard output by using the following method of System class:

```
System.setOut(PrintStream p);
```

**PrintStream** can be used for character output to a text file. Below program creates the file A.txt and writes to the file using System.out.println(

```
// Java program to demonstrate redirection in System.out.println()
import java.io.*;

public class SystemFact
{
    public static void main(String arr[]) throws FileNotFoundException
    {
        // Creating a File object that represents the disk file.
        PrintStream o = new PrintStream(new File("A.txt"));

        // Store current System.out before assigning a new value
        PrintStream console = System.out;

        // Assign o to output stream
        System.setOut(o);
        System.out.println("This will be written to the text file");

        // Use stored value for output stream
        System.setOut(console);
        System.out.println("This will be written on the console!");
    }
}
```

In very similar fashion we can use System.out.println() to write to a Socket's OutputStream as well.

This article is contributed by **Ashutosh Kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Copying file using FileStreams in Java

We can copy a file from one location to another using FileInputStream and FileOutputStream classes in Java.
For this we have to import some specific classes of java.io package. So for instance let us include the entire package with statement import java.io.*;

The main logic of copying file is to read the file associated to FileInputStream variable and write the read contents into the file associated with FileOutputStream variable.

**Methods used in the program**

1. **int read()**; Reads a byte of data. Present in FileInputStream. Other versions of this method : int read(byte[] bytearray) and int read(byte[] bytearray, int offset, int length)
2. **void write(int b)** : Writes a byte of data. Present in FileOutputStream. Other versions of this method : void write(byte[] bytearray) and void write(byte[] bytearray, int offset, int length);

```
/* Program to copy a src file to destination.
   The name of src file and dest file must be
   provided using command line arguments where
   args[0] is the name of source file and
   args[1] is name of destination file */

import java.io.*;
class src2dest
{
    public static void main(String args[])
    throws FileNotFoundException,IOException
    {
        /* If file doesnot exist FileInputStream throws
```

```
           FileNotFoundException and read() write() throws
           IOException if I/O error occurs */
        FileInputStream fis = new FileInputStream(args[0]);

        /* assuming that the file exists and need not to be
           checked */
        FileOutputStream fos = new FileOutputStream(args[1]);

        int b;
        while  ((b=fis.read()) != -1)
            fos.write(b);

        /* read() will readonly next int so we used while
           loop here in order to read upto end of file and
           keep writing the read int into dest file */
        fis.close();
        fos.close();
    }
}
```

This article is contributed by **Parul Dang**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Iterators in Java

Iterators are used in Collection framework in Java to retrieve elements one by one. There are three iterators.

**Enumeration :**

It is a interface used to get elements of legacy collections(Vector, Hashtable). Enumeration is the first iterator present from JDK 1.0, rests are included in JDK 1.2 with more functionality. Enumerations are also used to specify the input streams to a *SequenceInputStream*. We can create Enumeration object by calling *elements()* method of vector class on any vector object

```
// Here "v" is an Vector class object. e is of
// type Enumeration interface and refers to "v"
Enumeration e = v.elements();
```

There are **two** methods in Enumeration interface namely :

```
// Tests if this enumeration contains more elements
public boolean hasMoreElements();

// Returns the next element of this enumeration
// It throws NoSuchElementException
// if no more element present
public Object nextElement();
```

```
// Java program to demonstrate Enumeration
import java.util.Enumeration;
import java.util.Vector;

public class Test
{
    public static void main(String[] args)
    {
        // Create a vector and print its contents
        Vector v = new Vector();
        for (int i = 0; i < 10; i++)
            v.addElement(i);
        System.out.println(v);

        // At beginning e(cursor) will point to
        // index just before the first element in v
        Enumeration e = v.elements();

        // Checking the next element availability
        while (e.hasMoreElements())
        {
            // moving cursor to next element
            int i = (Integer)e.nextElement();

            System.out.print(i + " ");
        }
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
```

**Limitations of Enumeration :**

- Enumeration is for **legacy** classes(Vector, Hashtable) only. Hence it is not a universal iterator.
- Remove operations can't be performed using Enumeration.
- Only forward direction iterating is possible.

**Iterator:**

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.
Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

Iterator object can be created by calling *iterator()* method present in Collection interface.

```
// Here "c" is any Collection object. itr is of
// type Iterator interface and refers to "c"
Iterator itr = c.iterator();
```

Iterator interface defines **three** methods:

```
// Returns true if the iteration has more elements
public boolean hasNext();

// Returns the next element in the iteration
// It throws NoSuchElementException if no more
// element present
public Object next();

// Remove the next element in the iteration
// This method can be called only once per call
// to next()
public void remove();
```

*remove()* method can throw two exceptions

- *UnsupportedOperationException :* If the remove operation is not supported by this iterator
- *IllegalStateException :* If the next method has not yet been called, or the remove method has already been called after the last call to the next method

```
// Java program to demonstrate Iterator
import java.util.ArrayList;
import java.util.Iterator;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();

        for (int i = 0; i < 10; i++)
            al.add(i);

        System.out.println(al);

        // at beginning itr(cursor) will point to
        // index just before the first element in al
        Iterator itr = al.iterator();

        // checking the next element availabilty
        while (itr.hasNext())
        {
            //  moving cursor to next element
            int i = (Integer)itr.next();

            // getting even elements one by one
            System.out.print(i + " ");

            // Removing odd elements
            if (i % 2 != 0)
               itr.remove();
        }
        System.out.println();
        System.out.println(al);
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[0, 2, 4, 6, 8]
```

**Limitations of Iterator :**

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

**ListIterator:**

It is only applicable for List collection implemented classes like arraylist, linkedlist etc. It provides bi-directional iteration.
ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling *listIterator()* method present in List interface.

```
// Here "l" is any List object, ltr is of type
// ListIterator interface and refers to "l"
ListIterator ltr = l.listIterator();
```

ListIterator interface extends Iterator interface. So all three methods of Iterator interface are available for ListIterator. In addition there are **six** more methods.

```
// Forward direction

// Returns true if the iteration has more elements
public boolean hasNext();

// same as next() method of Iterator
public Object next();

// Returns the next element index
// or list size if the list iterator
// is at the end of the list
public int nextIndex();

// Backward direction

// Returns true if the iteration has more elements
// while traversing backward
public boolean hasPrevious();

// Returns the previous element in the iteration
// and can throws NoSuchElementException
// if no more element present
public Object previous();

// Returns the previous element index
//  or -1 if the list iterator is at the
// beginning of the list
```

```
public int previousIndex();

// Other Methods

// same as remove() method of Iterator
public void remove();

// Replaces the last element returned by
// next() or previous() with the specified element
public void set(Object obj);

// Inserts the specified element into the list at
// position before the element that would be returned
// by next(),
public void add(Object obj);
```

Clearly the three methods that *ListIterator* inherits from Iterator (*hasNext()*, *next()*, and *remove()*) do exactly the same thing in both interfaces. The *hasPrevious()* and the previous operations are exact analogues of *hasNext()* and *next()*. The former operations refer to the element before the (implicit) cursor, whereas the latter refer to the element after the cursor. The previous operation moves the cursor backward, whereas next moves it forward.

ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to *previous()* and the element that would be returned by a call to *next()*

*set()* method can throw four exceptions

- *UnsupportedOperationException* – if the set operation is not supported by this list iterator
- *ClassCastException :* If the class of the specified element prevents it from being added to this list
- *IllegalArgumentException :* If some aspect of the specified element prevents it from being added to this list
- *IllegalStateException :* If neither next nor previous have been called, or remove or add have been called after the last call to next or previous

*add()* method can throw three exceptions

- *UnsupportedOperationException :* If the add method is not supported by this list iterator
- *ClassCastException :* If the class of the specified element prevents it from being added to this list
- *IllegalArgumentException :* If some aspect of this element prevents it from being added to this list

```
// Java program to demonstrate ListIterator
import java.util.ArrayList;
import java.util.ListIterator;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList al = new ArrayList();
        for (int i = 0; i < 10; i++)
            al.add(i);

        System.out.println(al);

        // at beginning ltr(cursor) will point to
        // index just before the first element in al
        ListIterator ltr = al.listIterator();

        // checking the next element availabilty
        while (ltr.hasNext())
        {
            //  moving cursor to next element
            int i = (Integer)ltr.next();

            // getting even elements one by one
            System.out.print(i + " ");

            // Changing even numbers to odd and
            // adding modified number again in
            // iterator
            if (i%2==0)
            {
                i++;  // Change to odd
                ltr.set(i);  // set method to change value
                ltr.add(i);  // to add
            }
        }
        System.out.println();
        System.out.println(al);
    }
}
```

Output:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
0 1 2 3 4 5 6 7 8 9
[1, 1, 1, 3, 3, 3, 5, 5, 5, 7, 7, 7, 9, 9, 9]
```

**Limitations of ListIterator :** It is the most powerful iterator but it is only applicable for List implemented classes, so it is not a universal iterator.

**Important Common Points**

**1 :** Please note that initially any iterator reference will point to the index just before the index of first element in a collection.
**2 :** We don't create objects of Enumeration, Iterator, ListIterator because they are interfaces. We use methods like elements(), iterator(), listIterator() to create objects. These methods have anonymous Inner classes that extends respective interfaces and return this class object. This can be verified by below code. For more on inner class refer

```
// Java program to demonstrate iterators references
import java.util.Enumeration;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Vector;

public class Test
{
    public static void main(String[] args)
    {
        Vector v = new Vector();

        // Create three iterators
        Enumeration e = v.elements();
        Iterator  itr = v.iterator();
        ListIterator ltr = v.listIterator();
        // Print class names of iterators
        System.out.println(e.getClass().getName());
        System.out.println(itr.getClass().getName());
        System.out.println(ltr.getClass().getName());
    }
}
```

Output:

```
java.util.Vector$1
java.util.Vector$Itr
java.util.Vector$ListItr
```

The **$** symbol in reference class name is a proof that concept of inner classes is used and these class objects are created.

**Related Articles:**

Iterator vs Foreach In Java

Retrieving Elements from Collection in Java (For-each, Iterator, ListIterator & EnumerationIterator)

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Collections in Java

A Collection is a group of individual objects represented as a single unit. Java provides Collection Framework which defines several classes and interfaces to represent a group of objects as a single unit.

The Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are two main root interfaces of Java collection classes.

**Need for Collection Framework :**

Before Collection Framework (or before JDK 1.2) was introduced, the standard methods for grouping Java objects (or collections) were array or Vector or Hashtable. All three of these collections had no common interface.
For example, if we want to access elements of array, vector or Hashtable. All these three have different methods and syntax for accessing members:

```
// Java program to show whey collection framework was needed
import java.io.*;
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        // Creating instances of array, vector and hashtable
        int arr[] = new int[] {1, 2, 3, 4};
        Vector<Integer> v = new Vector();
        Hashtable<Integer, String> h = new Hashtable();
        v.addElement(1);
        v.addElement(2);
        h.put(1,"geeks");
        h.put(2,"4geeks");

        // Array instance creation requires [], while Vector
        // and hastable require ()
        // Vector element insertion requires addElement(), but
        // hashtable element insertion requires put()

        // Accessing first element of array, vector and hashtable
        System.out.println(arr[0]);
        System.out.println(v.elementAt(0));
        System.out.println(h.get(1));

        // Array elements are accessed using [], vector elements
        // using elementAt() and hashtable elements using get()
    }
}
```

Output:

```
1
1
geek
```

As we can see, none of the collections (Array, Vector or Hashtable) implements a standard member access interface. So, it was very difficult for programmers to write algorithm that can work for all kind of collections.
Another drawback is that, most of the 'Vector' methods are final. So, we cannot extend 'Vector' class to implement a similar kind of collection.
***Java developers decided to come up with a common interface to deal with the above mentioned problems and introduced Collection Framework***, they introduced collections in JDK 1.2 and changed the legacy Vector and Hashtable to conform to the collection framework.

**Advantages of Collection Framework:**

1. Consistent API : The API has basic set of interfaces like Collection, Set, List, or Map. All those classes (such as ArrayList, LinkedList, Vector etc) which implements, these interfaces have some common set of methods.
2. Reduces programming effort: The programmer need not to worry about design of Collection rather than he can focus on its best use in his program.
3. Increases program speed and quality: Increases performance by providing high-performance implementations of useful data structures and algorithms.

**Hierarchy of Collection Framework**

```
          Collection            Map
       /    /   \     \          |
      /    /     \     \         |
   Set   List   Queue  Dequeue   SortedMap
    /
   /
 SortedSet
          Core Interfaces in Collections

Note that this diagram shows only core interfaces.
```

```
Collection : Root interface with basic methods like add(), remove(),
           contains(), isEmpty(), addAll(), ... etc.

Set : Doesn't allow duplicates. Example implementations of Set
      interface are HashSet (Hashing based) and TreeSet (balanced
```

BST based). Note that TreeSet implements **SortedSet**.

**List :** Can contain duplicates and elements are ordered. Example implementations are LinkedList (linked list based) and ArrayList (dynamic array based)

**Queue :** Typically order elements in FIFO order except exceptions like PriorityQueue.

**Deque :** Elements can be inserted and removed at both ends. Allows both LIFO and FIFO.

**Map :** Contains Key value pairs. Doesn't allow duplicates. Example implementation are HashMap and TreeMap.
TreeMap implements **SortedMap**.

The difference between Set and Map interface is, in Set we have only keys, but in Map, we have key value pairs.

collectionjava

Image Source : https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

We will soon be discussing examples of interface implementations.

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java
Java-Collections

# How to use Iterator in Java?

'Iterator' is an interface which belongs to collection framework. It allows us to traverse the collection, access the data element and remove the data elements of the collection.

**java.util** package has **public interface Iterator** and contains three methods:

1. **boolean hasNext()**: It returns true if Iterator has more element to iterate.

2. **Object next()**: It returns the next element in the collection until the hasNext()method return true. This method throws 'NoSuchElementException' if there is no next element.

3.**void remove()**: It removes the current element in the collection. This method throws 'IllegalStateException' if this function is called before next( ) is invoked.

```
// Java code to illustrate the use of iterator
import java.io.*;
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        ArrayList<String> list = new ArrayList<String>();

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");
        list.add("E");

        // Iterator to traverse the list
        Iterator iterator = list.iterator();

        System.out.println("List elements : ");

        while (iterator.hasNext())
            System.out.print(iterator.next()+ " ");

        System.out.println();
    }
}
```

Output:

```
List elements :
A B C D E
```

'ListIterator' in Java is an Iterator which allows users to traverse Collection in both direction. It contains the following methods:

1. **void add(Object object)**: It inserts object in front of the element that is returned by the next( ) function.

2. **boolean hasNext( )**: It returns true if the list has a next element.

3. **boolean hasPrevious( )**: It returns true if the list has a previous element.

4. **Object next( )**: It returns the next element of the list. It throws 'NoSuchElementException' if there is no next element in the list.

5. **Object previous( )**: It returns the previous element of the list. It throws 'NoSuchElementException' if there is no previous element.

6. **void remove( )**: It removes the current element from the list. It throws 'IllegalStateException' if this function is called before next( ) or previous( ) is invoked.

```
// Java code to illustrate the use of ListIterator
import java.io.*;
import java.util.*;

class Test
{
    public static void main (String[] args)
    {
        ArrayList<String> list = new ArrayList<String>();

        list.add("A");
        list.add("B");
        list.add("C");
        list.add("D");
        list.add("E");

        // ListIterator to traverse the list
        ListIterator iterator = list.listIterator();

        // Traversing the list in forward direction
        System.out.println("Displaying list elements in forward direction : ");

        while (iterator.hasNext())
            System.out.print(iterator.next()+ " ");

        System.out.println();

        // Traversing the list in backward direction
        System.out.println("Displaying list elements in backward direction : ");

        while(iterator.hasPrevious())
            System.out.print(iterator.previous()+ " ");

        System.out.println();
    }
}
```

Output:

```
Displaying list elements in forward direction :
A B C D E
Displaying list elements in backward direction :
E D C B A
```

**Related Articles:**

- Iterators in Java
- Iterator vs Foreach In Java
- Retrieving Elements from Collection in Java (For-each, Iterator, ListIterator & EnumerationIterator)

This article is contributed by Mehak Narang.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

# Iterator vs Foreach In Java

**Background :**

**Iterator** is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

```
// Iterating over collection 'c' using terator
for (Iterator i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());
```

**For each** loop is meant for traversing items in a collection.

```
// Iterating over collection 'c' using for-each
for (Element e: c)
    System.out.println(e);
```

We read the ':' used in for-each loop as "in". So loop reads as "for each element e in elements", here elements is the collection which stores Element type items.

**Note :** In Java 8 using lambda expressions we can simply replace for-each loop with

```
elements.forEach (e -> System.out.println(e) );
```

**Difference between the two traversals**

In for-each loop, we can't modify collection, it will throw a ConcurrentModificationException on the other hand with iterator we can modify collection.
Modifying a collection simply means removing an element or changing content of an item stored in the collection. This occurs because for-each loop implicitly creates an iterator but it is not exposed to the user thus we can't modify the items in the collections.

**When to use which traversal?**

- If we have to modify collection, we must use an Iterator.
- While using nested for loops it is better to use for-each loop, consider the below code for better understanding.

```
// Java program to demonstrate working of nested iterators
// may not work as expected and throw exception.
import java.util.*;

public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l = new LinkedList<Integer>();

        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(7);
        s.add(8);
        s.add(9);

        // Iterator to iterate over a Link List
        for (Iterator<Integer> itr1=l.iterator(); itr1.hasNext(); )
        {
            for (Iterator<Integer> itr2=s.iterator(); itr2.hasNext(); )
            {
                if (itr1.next() < itr2.next())
                {
                    System.out.println(itr1.next());
                }
            }
        }
    }
}
```

Output:

```
Exception in thread "main" java.util.NoSuchElementException
 at java.util.LinkedList$ListItr.next(LinkedList.java:888)
 at Main.main(Main.java:29)
```

The above code throws java.util.NoSuchElementException.

In the above code we are calling the next() method again and again for itr1 (i.e., for List l). Now we are advancing the iterator without even checking if it has any more elements left in the collection(in the inner loop), thus we are advancing the iterator more than the number of elements in the collection which leads to NoSuchElementException.

for-each loops are tailor made for nested loops. Replace the iterator code with the below code.

```
// Java program to demonstrate working of nested for-each
import java.util.*;
public class Main
{
    public static void main(String args[])
    {
        // Create a link list which stores integer elements
        List<Integer> l=new LinkedList<Integer>();

        // Now add elements to the Link List
        l.add(2);
        l.add(3);
        l.add(4);

        // Make another Link List which stores integer elements
        List<Integer> s=new LinkedList<Integer>();
        s.add(2);
        s.add(4);
        s.add(5);
        s.add(6);

        // Iterator to iterate over a Link List
        for (int a:l)
        {
            for (int b:s)
            {
                if (a<b)
                    System.out.print(a + " ");
            }
        }
    }
}
```

**Output:**

```
2 2 2 3 3 3 4 4
```

**Performance Analysis**

Traversing a collection using for-each loops or iterators give the same performance. Here, by performance we mean the time complexity of both these traversals.

If you iterate using the old styled C for loop then we might increase the time complexity drastically.

// Here l is List ,it can be ArrayList /LinkedList and n is size of the List

```
for (i=0;i<n;i++)
   System.out.println(l.get(i));
```

Here if the list l is an ArrayList then we can access it in O(1) time since it is allocated contiguous memory blocks (just like an array) i.e random access is possible. But if the collection is LinkedList, then random access is not possible since it is not allocated contiguous memory blocks, so in order to access a element we will have to traverse the link list till you get to the required index, thus the time taken in worst case to access an element will be O(n).

**Iterator and for-each loop are faster than simple for loop for collections with no random access, while in collections which allows random access there is no performance change with for-each loop/for loop/iterator.**

**Related Articles:**
Iterators in Java
Retrieving Elements from Collection in Java (For-each, Iterator, ListIterator & EnumerationIterator)

**References:**
https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html
https://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html
https://stackoverflow.com/questions/2113216/which-is-more-efficient-a-for-each-loop-or-an-iterator

This article is contributed by **Chirag Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Retrieving Elements from Collection in Java (For-each, Iterator, ListIterator & EnumerationIterator)

Prerequisite : Collection in Java

Following are the 4 ways to retrieve any elements from a collection object:

**For-each**

For each loop is meant for traversing items in a collection.

```
// Iterating over collection 'c' using for-each
for (Element e: c)
    System.out.println(e);
```

We read the ':' used in for-each loop as "in". So loop reads as "for each element e in elements", here elements is the collection which stores Element type items.

**Note :** In Java 8 using lambda expressions we can simply replace for-each loop with

```
elements.forEach (e -> System.out.println(e) );
```

**Iterator Interface**

Iterator is an interface provided by collection framework to traverse a collection and for a sequential access of items in the collection.

```
// Iterating over collection 'c' using terator
for (Iterator i = c.iterator(); i.hasNext(); )
    System.out.println(i.next());
```

It has 3 methods:

- *boolean hasNext():* This method returns true if the iterator has more elements.
- *elements next()*: This method returns the next elements in the iterator.
- *void remove()*: This method removes from the collection the last elements returned by the iterator.

```
// Java program to demonstrate working of iterators
import java.util.*;
public class IteratorDemo
{
    public static void main(String args[])
    {
        //create a Hashset to store strings
        HashSet<String> hs = new HashSet<String>() ;

        // store some string elements
        hs.add("India");
        hs.add ("America");
        hs.add("Japan");

        // Add an Iterator to hs.
        Iterator it = hs.iterator();

        // Display element by element using Iterator
        while (it.hasNext())
            System.out.println(it.next());
    }
}
```

Output:

```
America
Japan
India
```

Refer Iterator vs For-each for differences between iterator and for-each.

**ListIterator Interface**

*ListIterator*is an interface that contains methods to retrieve the elements from a collection object, both in **forward and reverse directions**. This iterator is for list based collections.

It has following important methods:

- *booleanhasNext()*: This returns true if the ListIterator has more elements when traversing the list in the forward direction.
- *booleanhasPerivous()*: This returns true if the ListIterator has more elements when traversing the list in the reverse direction.
- *element next()*: This returns the next element in the list.
- *element previous()*:This returns the previous element in the list.
- *void remove()*: This removes from the list the last elements that was returned by the next() or previous() methods.

- *int nextIndex()* Returns the index of the element that would be returned by a subsequent call to next(). (Returns list size if the list iterator is at the end of the list.)
- *int previousIndex()* Returns the index of the element that would be returned by a subsequent call to previous(). (Returns -1 if the list iterator is at the beginning of the list.)

Source: ListIterator Oracle Docs

**How is Iterator different from ListIterator?**

Iterator can retrieve the elements only in forward direction. But ListIterator can retrieve the elements in forward and reverse direction also. So ListIterator is preferred to Iterator.

Using ListItarator, we can get iterator's current position

Since ListIterator can access elements in both directions and supports additional operators, ListIterator cannot be applied on Set (e.g., HashSet and TreeSet. See this). However, we can use LisIterator with vector and list (e.g. ArrayList ).

```
// Java program to demonstrate working of ListIterator
import java. util.* ;

class Test
{
    public static void main(String args[])
    {
        // take a vector to store Integer objects
        Vector<Integer> v = new Vector<Integer>();

        // Adding Elements to Vector
        v.add(10);
        v.add(20);
        v.add(30);

        // Creating a ListIterator
        ListIterator lit = v.listIterator();
        System.out.println("In Forward direction:");

        while (lit.hasNext())
            System.out.print(lit.next()+" ") ;

        System.out.print("\n\nIn backward direction:\n") ;
        while (lit.hasPrevious())
            System.out.print(lit.previous()+" ");
    }
}
```

Output :

```
In Forward direction:
10 20 30

In backward direction:
30 20 10
```

**EnumerationIterator Interface**

The interface is useful to retrieve one by one the element. This iterator is based on data from Enumeration and has methods:

- *booleanhasMoreElements()*: This method tests if the Enumeration has any more elements or not .
- *element nextElement()*: This returns the next element that is available in elements that is available in Enumeration

**What is the difference between Iterator and Enumeration?**

Iterator has an option to remove elements from the collection which is not available in Enumeration. Also, Iterator has methods whose names are easy to follow and Enumeration methods are difficult to remember.

```
import java.util.Vector;
import java.util.Enumeration;
public class Test
{
    public static void main(String args[])
    {
        Vector dayNames = new Vector();
        dayNames.add("Sunday");
        dayNames.add("Monday");
        dayNames.add("Tuesday");
        dayNames.add("Wednesday");
        dayNames.add("Thursday");
        dayNames.add("Friday");
        dayNames.add("Saturday");

        // Creating enumeration
        Enumeration days = dayNames.elements();

        // Retrieving elements of enumeration
        while (days.hasMoreElements())
            System.out.println(days.nextElement());
    }
}
```

**Output:**

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

**Related Articles:**
Iterators in Java
Iterator vs Foreach In Java

This article is contributed by **Nishant Sharma.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Java
Java

## Set in Java

- Set is an interface which extends Collection. It is an unordered collection of objects in which duplicate values cannot be stored.
- Basically, Set is implemented by HashSet, LinkedSet or TreeSet (sorted representation).
- Set has various methods to add, remove clear, size, etc to enhance the usage of this interface

```
// Java code for adding elements in Set
import java.util.*;
public class Set_example
{
    public static void main(String[] args)
    {
        // Set deonstration using HashSet
        Set<String> hash_Set = new HashSet<String>();
        hash_Set.add("Geeks");
        hash_Set.add("For");
        hash_Set.add("Geeks");
        hash_Set.add("Example");
        hash_Set.add("Set");
        System.out.print("Set output without the duplicates");

        System.out.println(hash_Set);

        // Set deonstration using TreeSet
        System.out.print("Sorted Set after passing into TreeSet");
        Set<String> tree_Set = new TreeSet<String>(hash_Set);
        System.out.println(tree_Set);
    }
}
```

(Please note that we have entered a duplicate entity but it is not displayed in the output. Also, we can directly sort the entries by passing the unordered Set in as the parameter of TreeSet).

Output:

```
Set output without the duplicates[Geeks, Example, For, Set]
Sorted Set after passing into TreeSet[Example, For, Geeks, Set]
```

Note: As we can see the duplicate entry “Geeks” is ignored in the final output, Set interface doesn’t allow duplicate entries.

Now we will see some of the basic operations on the Set i.e. Union, Intersection and Difference.

Let’s take an example of two integer Sets:

- [1, 3, 2, 4, 8, 9, 0]
- [1, 3, 7, 5, 4, 0, 7, 5]

**Union**
In this, we could simply add one Set with other. Since the Set will itself not allow any duplicate entries, we need not take care of the common values.

Expected Output:

```
Union : [0, 1, 2, 3, 4, 5, 7, 8, 9]
```

**Intersection**
We just need to retain the common values from both Sets.

Expected Output:

```
Intersection : [0, 1, 3, 4]
```

**Difference**
We just need to remove tall the values of one Set from the other.

Expected Output:

```
Difference : [2, 8, 9]
```

```
// Java code for demonstrating union, intersection and difference
// on Set
import java.util.*;
public class Set_example
{
    public static void main(String args[])
    {
        Set<Integer> a = new HashSet<Integer>();
        a.addAll(Arrays.asList(new Integer[] {1, 3, 2, 4, 8, 9, 0}));
        Set<Integer> b = new HashSet<Integer>();
        b.addAll(Arrays.asList(new Integer[] {1, 3, 7, 5, 4, 0, 7, 5}));

        // To find union
        Set<Integer> union = new HashSet<Integer>(a);
        union.addAll(b);
        System.out.print("Union of the two Set");
        System.out.println(union);

        // To find intersection
        Set<Integer> intersection = new HashSet<Integer>(a);
        intersection.retainAll(b);
        System.out.print("Intersection of the two Set");
        System.out.println(intersection);

        // To find the symmetric difference
        Set<Integer> difference = new HashSet<Integer>(a);
        difference.removeAll(b);
        System.out.print("Difference of the two Set");
        System.out.println(difference);
    }
}
```

Output:

```
Union of the two Set[0, 1, 2, 3, 4, 5, 7, 8, 9]
Intersection of the two Set[0, 1, 3, 4]
Difference of the two Set[2, 8, 9]
```

This article is contributed by **Pranjal Mathur**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# List Interface in Java with Examples

Java.util.List is a child interface of Collection.

listinterfacejava

List is an ordered collection of objects in which duplicate values can be stored. Since List preserves the insertion order it allows positional access and insertion of elements. List Interface is implemented by ArrayList, LinkedList, Vector and Stack classes.

**Creating List Objects:**
List is an interface, we can create instance of List in following ways:

```
List a = new ArrayList();
List b = new LinkedList();
List c = new Vector();
List d = new Stack();
```

**Generic List Object:**
After the introduction of Generics in Java 1.5, it is possible to restrict the type of object that can be stored in the List. We can declare type safe List in following way:

```
// Obj is type of object to be stored in List.
List<Obj> list = new List<Obj> ();
```

**Operations on List:**
List Interface extends Collection, hence it supports all the operations of Collection Interface and along with following operations:

1. **Positional Access:**
   List allows add, remove, get and set operations based on numerical positions of elements in List. List provides following methods for these operations:
   - **void add(int index,Object O):** This method adds given element at specified index.
   - **boolean addAll(int index, Collection c):** This method adds all elements from specified collection to list. First element gets inserted at given index. If there is already an element at that position, that element and other subsequent elements(if any) are shifted to the right by increasing their index.
   - **Object remove(int index):** This method removes an element from the specified index. It shifts subsequent elements(if any) to left and decreases their indexes by 1.
   - **Object get(int index):** This method returns element at the specified index.
   - **Object set(int index, Object new):** This method replaces element at given index with new element. This function returns the element which was just replaced by new element.

```
// Java program to demonstrate positional access
// operations on List interface
import java.util.*;

public class ListDemo
{
    public static void main (String[] args)
    {
        // Let us create a list
        List l1 = new ArrayList();
        l1.add(0, 1);  // adds 1 at 0 index
        l1.add(1, 2);  // adds 2 at 1 index
        System.out.println(l1);  // [1, 2]

        // Let us create another list
        List l2 = new ArrayList();
        l2.add(1);
        l2.add(2);
        l2.add(3);

        // will add list l2 from 1 index
        l1.addAll(1, l2);
        System.out.println(l1);

        l1.remove(1);     // remove element from index 1
        System.out.println(l1); // [1, 2, 3, 2]

        // prints element at index 3
        System.out.println(l1.get(3));

        l1.set(0, 5);   // replace 0th element with 5
        System.out.println(l1);  // [5, 2, 3, 2]
    }
}
```

Output :

```
[1, 2]
[1, 1, 2, 3, 2]
[1, 2, 3, 2]
2
```

```
[5, 2, 3, 2]
```

2. **Search:**
   List provides methods to search element and returns its numeric position. Following two methods are supported by List for this operation:
   - **int indexOf(Object o):** This method returns first occurrence of given element or -1 if element is not present in list.
   - **int lastIndexOf(Object o):** This method returns the last occurrence of given element or -1 if element is not present in list.

```
// Java program to demonstrate search
// operations on List interface
import java.util.*;

public class ListDemo
{
    public static void main(String[] args)
    {
        // type safe array list, stores only string
        List<String> l = new ArrayList<String>(5);
        l.add("Geeks");
        l.add("for");
        l.add("Geeks");

        // Using indexOf() and lastIndexOf()
        System.out.println("first index of Geeks:" +
                          l.indexOf("Geeks"));
        System.out.println("last index of Geeks:" +
                       l.lastIndexOf("Geeks"));
        System.out.println("Index of element not present : " +
                        l.indexOf("Hello"));
    }
}
```

   Output :

```
first index of Geeks:0
last index of Geeks:2
Index of element not present : -1
```

3. **Iteration:**
   ListIterator(extends Iterator) is used to iterate over List element. List iterator is bidirectional iterator. For more details about ListIterator refer Iterators in Java.

4. **Range-view:**
   List Interface provides method to get List view of the portion of given List between two indices. Following is the method supported by List for range view operation.
   - **List subList(int fromIndex,int toIndex):**This method returns List view of specified List between fromIndex(inclusive) and toIndex(exclusive).

```
// Java program to demonstrate subList operation
// on List interface.
import java.util.*;
public class ListDemo
{
    public static void main (String[] args)
    {
        // Type safe array list, stores only string
        List<String> l = new ArrayList<String>(5);

        l.add("GeeksforGeeks");
        l.add("Practice");
        l.add("GeeksQuiz");
        l.add("IDE");
        l.add("Courses");

        List<String> range = new ArrayList<String>();

        // return List between 2nd(including)
        // and 4th element(excluding)
        range = l.subList(2, 4);

        System.out.println(range);  // [GeeksQuiz, IDE]
    }
}
```

   Output :

```
[GeeksQuiz, IDE]
```

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# HashSet in Java

**HashSet:**

- Implements Set Interface.
- Underlying data structure for HashSet is hashtable.
- As it implements the Set Interface, duplicate values are not allowed.
- Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
- NULL elements are allowed in HashSet.
- HashSet also implements Searlizable and Cloneable interfaces.

**Constructors in HashSet:**

```
HashSet h = new HashSet();
Default initial capacity is 16 and default load factor is 0.75.

HashSet h = new HashSet(int initialCapacity);
default loadFactor of 0.75

HashSet h = new HashSet(int initialCapacity, float loadFactor);

HashSet h = new HashSet(Collection C);
```

**What is initial capacity and load factor?**

The initial capacity means the number of buckets when hashtable (HashSet internally uses hashtable data structure) is created. Number of buckets will be automatically increased if the current size gets full.
The load factor is a measure of how full the HashSet is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the hash table is rehashed (that is, internal data structures are rebuilt) so that the hash table has approximately twice the number of buckets.

```
              Number of stored elements in the table
   load factor = -----------------------------------------
                        Size of the hash table
```

**E.g.** If internal capacity is 16 and load factor is 0.75 then, number of buckets will automatically get increased when table has 12 elements in it.

**Effect on performance:**
Load factor and initial capacity are two main factors that affect the performance of HashSet operations. Load factor of 0.75 provides very effective performance as respect to time and space complexity. If we increase the load factor value more than that then memory overhead will be reduced (because it will decrease internal rebuilding operation) but, it will affect the add and search operation in hashtable. To reduce the rehashing operation we should choose initial capacity wisely. If initial capacity is greater than the maximum number of entries divided by the load factor, no rehash operation will ever occur.

**Important Methods in HashSet:**

1. **boolean add(E e)** : add the specified element if it is not present, if it is present then return false.
2. **void clear()** : removes all the elements from set.
3. **boolean contains(Object o)** : return true if element is present in set.
4. **boolean remove(Object o)** : remove the element if it is present in set.
5. **Iterator iterator()** : return an iterator over the element in the set.

**Sample Program:**

```
// Java program to demonstrate working of HashSet
import java.util.*;

class Test
{
    public static void main(String[]args)
    {
        HashSet<String> h = new HashSet<String>();

        // adding into HashSet
        h.add("India");
        h.add("Australia");
        h.add("South Africa");
        h.add("India");// adding duplicate elements

        // printing HashSet
        System.out.println(h);
        System.out.println("List contains India or not:" +
                           h.contains("India"));

        // Removing an item
        h.remove("Australia");
        System.out.println("List after removing Australia:"+h);

        // Iterating over hash set items
        System.out.println("Iterating over list:");
        Iterator<String> i = h.iterator();
        while (i.hasNext())
            System.out.println(i.next());
    }
}
```

**Output of the above program:**

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

**How HashSet internally work?**
All the classes of Set interface internally backed up by Map. HashSet uses HashMap for storing its object internally. You must be wondering that to enter a value in HashMap we need a key-value pair, but in HashSet we are passing only one value.

**Then how is it storing in HashMap?**
Actually the value we insert in HashSet acts as key to the map Object and for its value java uses a constant variable. So in key-value pair all the keys will have same value.

**If we look at the implementation of HashSet in java doc, it is something like this;**

```
private transient HashMap map;

// Constructor - 1
// All the constructors are internally creating HashMap Object.
public HashSet()
{
    // Creating internally backing HashMap object
    map = new HashMap();
}

// Constructor - 2
public HashSet(int initialCapacity)
{
    // Creating internally backing HashMap object
    map = new HashMap(initialCapacity);
}

// Dummy value to associate with an Object in Map
private static final Object PRESENT = new Object();
```

If we look at add() method of HashSet class:

```
public boolean add(E e)
{
   return map.put(e, PRESENT) == null;
}
```

We can notice that, add() method of HashSet class internally calls put() method of backing HashMap object by passing the element you have specified as a key and constant “PRESENT” as its value.

remove() method also works in the same manner. It internally calls remove method of Map interface.

```
public boolean remove(Object o)
{
  return map.remove(o) == PRESENT;
}
```

**Time Complexity of HashSet Operations:**

The underlying data structure for HashSet is hashtable. So amortize (average or usual case) time complexity for add, remove and look-up (contains method) operation of HashSet takes **O(1)** time.

**References:**

https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html

This article is contributed by **Dharmesh Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Articles Java

# LinkedList in java

In Java, LinkedList class implements the list interface.

This class consists of the following methods :

1. **boolean add(Object element)** : It appends the element to the end of the list.

2. **void add(int index, Object element)**: It inserts the element at the position ‘index’ in the list.

3. **void addFirst(Object element)** : It inserts the element at the beginning of the list.

4. **void addLast(Object element)** : It appends the element at the end of the list.

5. **boolean contains(Object element)** : It returns true if the element is present in the list.

6. **Object get(int index)** : It returns the element at the position ‘index’ in the list. It throws ‘IndexOutOfBoundsException’ if the index is out of range of the list.

7. **int indexOf(Object element)** : If element is found, it returns the index of the first occurrence of the element. Else, it returns -1.

8. **Object remove(int index)** : It removes the element at the position ‘index’ in this list. It throws ‘NoSuchElementException’ if the list is empty.

9. **int size()** : It returns the number of elements in this list.

10. **void clear()** : It removes all of the elements from the list.

```
// Java code for Linked List implementation

import java.util.*;

public class Test
{
    public static void main(String args[])
    {
        // Creating object of class linked list
        LinkedList<String> object = new LinkedList<String>();

        // Adding elements to the linked list
        object.add("A");
        object.add("B");
        object.addLast("C");
        object.addFirst("D");
        object.add(2, "E");
        object.add("F");
        object.add("G");
        System.out.println("Linked list : " + object);

        // Removing elements from the linked list
        object.remove("B");
        object.remove(3);
        object.removeFirst();
        object.removeLast();
        System.out.println("Linked list after deletion: " + object);

        // Finding elements in the linked list
        boolean status = object.contains("E");

        if(status)
            System.out.println("List contains the element 'E' ");
        else
            System.out.println("List doesn't contain the element 'E'");

        // Number of elements in the linked list
        int size = object.size();
        System.out.println("Size of linked list = " + size);

        // Get and set elements from linked list
        Object element = object.get(2);
        System.out.println("Element returned by get() : " + element);
        object.set(2, "Y");
        System.out.println("Linked list after change : " + object);
    }
}
```

Output :

```
Linked list : [D, A, E, B, C, F, G]
Linked list after deletion: [A, E, F]
```

```
List contains the element 'E'
Size of linked list = 3
Element returned by get() : F
Linked list after change : [A, E, Y]
```

This article is contributed by Mehak Narang.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java Linked List

# ArrayList vs LinkedList in Java

Two popular lists in Java are:

1. ArrayList:-Implemented with the concept of dynamic array.

```
ArrayList<Type> arrL = new ArrayList<Type>();

Here Type is the data type of elements in ArrayList
to be created
```

2. LinkedList:-Implemented with the concept of doubly linked list.

```
LinkedList<Type> linkL = new LinkedList<Type>();

Here Type is the data type of elements in LinkedList
to be created
```

Comparision between ArrayList and LinkedList:-

1. Insertions are easy and fast in LinkedList as compared to ArrayList because there is no risk of resizing array and copying content to new array if array gets full which makes adding into ArrayList of O(n) in worst case, while adding is O(1) operation in LinkedList in Java. ArrayList also needs to be update its index if you insert something anywhere except at the end of array.
2. Removal also better in LinkedList than ArrayList due to same reasons as insertion.
3. LinkedList has more memory overhead than ArrayList because in ArrayList each index only holds actual object (data) but in case of LinkedList each node holds both data and address of next and previous node.
4. Both LinkedList and ArrayList require O(n) time to find if an element is present or not. However we can do Binary Search on ArrayList if it is sorted and therefore can search in O(Log n) time.

```
// Java program to demonstrate difference between ArrayList and
// LinkedList.
import java.util.ArrayList;
import java.util.LinkedList;

public class ArrayListLinkedListExample
{
    public static void main(String[] args)
    {
        ArrayList<String> arrlistobj = new ArrayList<String>();
        arrlistobj.add("0. Practice.GeeksforGeeks.org");
        arrlistobj.add("1. Quiz.GeeksforGeeks.org");
        arrlistobj.add("2. Code.GeeksforGeeks.org");
        arrlistobj.remove(1);  // Remove value at index 2
        System.out.println("ArrayList object output :" +  arrlistobj);

        // Checking if an element is present.
        if (arrlistobj.contains("2. Code.GeeksforGeeks.org"))
            System.out.println("Found");
        else
            System.out.println("Not found");


        LinkedList llobj = new LinkedList();
        llobj.add("0. Practice.GeeksforGeeks.org");
        llobj.add("1. Quiz.GeeksforGeeks.org");
        llobj.add("2. Code.GeeksforGeeks.org");
        llobj.remove("1. Quiz.GeeksforGeeks.org");
        System.out.println("LinkedList object output :" + llobj);

        // Checking if an element is present.
        if (llobj.contains("2. Code.GeeksforGeeks.org"))
            System.out.println("Found");
        else
            System.out.println("Not found");
    }
}
```

**Output:**

```
ArrayList object output :[0. Practice.GeeksforGeeks.org, 2. Code.GeeksforGeeks.org]
Found
LinkedList object output :[0. Practice.GeeksforGeeks.org, 2. Code.GeeksforGeeks.org]
Found
```

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

# Stack Class in Java

Java provides an inbuilt object type called **Stack**. It is a collection that is based on the last in first out (LIFO) principle. On Creation, a stack is empty.

It extends **Vector** class with five methods that allow a vector to be treated as a stack. The five methods are:

1. **Object push(Object element)** : Pushes an element on the top of the stack.

2. **Object pop()** : Removes and returns the top element of the stack. An 'EmptyStackException' exception is thrown if we call pop() when the invoking stack is empty.

3. **Object peek( )** : Returns the element on the top of the stack, but does not remove it.

4. **boolean empty()** : It returns true if nothing is on the top of the stack. Else, returns false.

5. **int search(Object element)** : It determines whether an object exists in the stack. If the element is found, it returns the position of the element from the top of the stack. Else, it returns -1.

```
// Java code for stack implementation

import java.io.*;
import java.util.*;

class Test
{
    // Pushing element on the top of the stack
    static void stack_push(Stack<Integer> stack)
    {
        for(int i = 0; i < 5; i++)
        {
            stack.push(i);
        }
    }

    // Popping element from the top of the stack
    static void stack_pop(Stack<Integer> stack)
    {
        System.out.println("Pop :");

        for(int i = 0; i < 5; i++)
        {
            Integer y = (Integer) stack.pop();
            System.out.println(y);
        }
    }

    // Displaying element on the top of the stack
    static void stack_peek(Stack<Integer> stack)
    {
        Integer element = (Integer) stack.peek();
        System.out.println("Element on stack top : " + element);
    }

    // Searching element in the stack
    static void stack_search(Stack<Integer> stack, int element)
    {
        Integer pos = (Integer) stack.search(element);

        if(pos == -1)
            System.out.println("Element not found");
        else
            System.out.println("Element is found at position " + pos);
    }


    public static void main (String[] args)
    {
        Stack<Integer> stack = new Stack<Integer>();

        stack_push(stack);
        stack_pop(stack);
        stack_push(stack);
        stack_peek(stack);
        stack_search(stack, 2);
        stack_search(stack, 6);
    }
}
```

Output :

```
Pop :
4
3
2
1
0
Element on stack top : 4
Element is found at position 3
Element not found
```

This article is contributed by Mehak Narang.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

# Differences between HashMap and HashTable in Java

HashMap and Hashtable store key/value pairs in a hash table. When using a Hashtable or HashMap, we specify an object that is used as a key, and the value that you want linked to that key. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

Sample Java code.

```
// A sample Java program to demonstrate HashMap and HashTable
import java.util.*;
import java.lang.*;
import java.io.*;

/* Name of the class has to be "Main" only if the class is public. */
class Ideone
{
    public static void main(String args[])
    {
        //----------hashtable -------------------------
        Hashtable<Integer,String> ht=new Hashtable<Integer,String>();
        ht.put(101," ajay");
        ht.put(101,"Vijay");
        ht.put(102,"Ravi");
        ht.put(103,"Rahul");
        System.out.println("-------------Hash table--------------");
        for (Map.Entry m:ht.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
        }

        //----------------hashmap--------------------------------
        HashMap<Integer,String> hm=new HashMap<Integer,String>();
        hm.put(100,"Amit");
        hm.put(104,"Amit");  // hash map allows duplicate values
        hm.put(101,"Vijay");
        hm.put(102,"Rahul");
        System.out.println("-----------Hash map-----------");
        for (Map.Entry m:hm.entrySet()) {
            System.out.println(m.getKey()+" "+m.getValue());
        }
    }
}
```

Output:

```
-------------Hash table--------------
103 Rahul
102 Ravi
101 Vijay
-----------Hash map-----------
100 Amit
101 Vijay
102 Rahul
104 Amit
```

**Hashmap vs Hashtable**
1. HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
2. HashMap allows one null key and multiple null values whereas Hashtable doesn't allow any null key or value.
3. HashMap is generally preferred over HashTable if thread synchronization is not needed

Why HashTable doesn't allow null and HashMap does?
To successfully store and retrieve objects from a HashTable, the objects used as keys must implement the hashCode method and the equals method. Since null is not an object, it can't implement these methods. HashMap is an advanced version and improvement on the Hashtable. HashMap was created later.

**Sources:**
http://docs.oracle.com/javase/6/docs/api/java/util/Hashtable.html:
http://qa.geeksforgeeks.org/558/differences-between-hashmap-hashtable-and-hashset-in-java?show=558#q558

This article is compiled by **Aditya Goel**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# HashMap and TreeMap in Java

HashMap and TreeMap are part of collection framework.

**HashMap**

java.util.HashMap class is a Hashing based implementation. In HashMap, we have a key and a value pair<Key, Value>.

```
 HashMap<K, V> hmap = new HashMap<K, V>();
```

Let us consider below example where we have to count occurrences of each integer in given array of integers.

```
Input: arr[] = {10, 3, 5, 10, 3, 5, 10};
Output: Frequency of 10 is 3
        Frequency of 3 is 2
        Frequency of 5 is 2
```

```
/* Java program to print frequencies of all elements using
   HashMap */
import java.util.*;

class Main
{
    // This function prints frequencies of all elements
    static void printFreq(int arr[])
    {
        // Creates an empty HashMap
        HashMap<Integer, Integer> hmap =
                     new HashMap<Integer, Integer>();

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++)
        {
            Integer c = hmap.get(arr[i]);

            // If this is first occurrence of element
            if (hmap.get(arr[i]) == null)
               hmap.put(arr[i], 1);

            // If elements already exists in hash map
            else
              hmap.put(arr[i], ++c);
        }

        // Print result
        for (Map.Entry m:hmap.entrySet())
          System.out.println("Frequency of " + m.getKey() +
                             " is " + m.getValue());
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 34, 5, 10, 3, 5, 10};
        printFreq(arr);
    }
}
```

Output:

```
Frequency of 34 is 1
Frequency of 3 is 1
Frequency of 5 is 2
Frequency of 10 is 3
```

**Key Points**

- HashMap does not maintain any order neither based on key nor on basis of value, If we want the keys to be maintained in a sorted order, we need to use TreeMap.
- **Complexity**: get/put/containsKey() operations are O(1) in average case but we can't guarantee that since it all depends on how much time does it take to compute the hash.

**Application:**
HashMap is basically an implementation of hashing. So wherever we need hashing with key value pairs, we can use HashMap. For example, in Web Applications username is stored as a key and user data is stored as a value in the HashMap, for faster retrieval of user data corresponding to a username.

**TreeMap**

TreeMap can be a bit handy when we only need to store unique elements in a sorted order. Java.util.TreeMap uses a red-black tree in the background which makes sure that there are no duplicates; additionally it also maintains the elements in a sorted order.

```
 TreeMap<K, V> hmap = new TreeMap<K, V>();
```

Below is TreeMap based implementation of same problem. This solution has more time complexity O(nLogn) compared to previous one which has O(n). The advantage of this method is, we get elements in sorted order.

```
/* Java program to print frequencies of all elements using
   TreeMap */
import java.util.*;

class Main
{
    // This function prints frequencies of all elements
    static void printFreq(int arr[])
    {
        // Creates an empty TreeMap
        TreeMap<Integer, Integer> tmap =
                     new TreeMap<Integer, Integer>();

        // Traverse through the given array
        for (int i = 0; i < arr.length; i++)
        {
            Integer c = tmap.get(arr[i]);

            // If this is first occurrence of element
            if (tmap.get(arr[i]) == null)
               tmap.put(arr[i], 1);

            // If elements already exists in hash map
            else
              tmap.put(arr[i], ++c);
        }

        // Print result
        for (Map.Entry m:tmap.entrySet())
          System.out.println("Frequency of " + m.getKey() +
                             " is " + m.getValue());
    }

    // Driver method to test above method
    public static void main (String[] args)
    {
        int arr[] = {10, 34, 5, 10, 3, 5, 10};
        printFreq(arr);
    }
}
```

Output:

```
Frequency of 3 is 1
Frequency of 5 is 2
Frequency of 10 is 3
Frequency of 34 is 1
```

**Key Points**

- For operations like add, remove, containsKey, time complexity is O(log n where n is number of elements present in TreeMap.

- TreeMap always keeps the elements in a sorted(increasing) order, while the elements in a HashMap have no order. TreeMap also provides some cool methods for first, last, floor and ceiling of keys.

**Overview:**

1. HashMap implements Map interface while TreeMap implements SortedMap interface. A Sorted Map interface is a child of Map.
2. HashMap implements Hashing, while TreeMap implements Red-Black Tree(a Self Balancing Binary Search Tree). Therefore all differences between Hashing and Balanced Binary Search Tree apply here.
3. Both HashMap and TreeMap have their counterparts HashSet and TreeSet. HashSet and TreeSet implement Set interface. In HashSet and TreeSet, we have only key, no value, these are mainly used to see presence/absence in a set. For above problem, we can't use HashSet (or TreeSet) as we can't store counts. An example problem where we would prefer HashSet (or TreeSet) over HashMap (or TreeMap) is to print all distinct elements in an array.

collection_interfaces

Link to the Image: https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

**Also see:**

- LinkedHashmap in Java

**References :**
https://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

This article is contributed by **Chirag Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

# ArrayList in Java

ArrayList is a part of collection framework and is present in java.util package. It provides us dynamic arrays in Java. Though, it may be slower than standard arrays but can be helpful in programs where lots of manipulation in the array is needed.

- ArrayList inherits AbstractList class and implements List interface.
- ArrayList is initialized by a size, however the size can increase if collection grows or shrunk if objects are removed from the collection.
- Java ArrayList allows us to randomly access the list.
- ArrayList can not be used for primitive types, like int, char, etc. We need a wrapper class for such cases (see this for details).
- ArrayList in Java can be seen as similar to vector in C++.

Let us look at the code to create generic ArrayList-

```
// Creating generic integer ArrayList
ArrayList<Integer> arrli = new ArrayList<Integer>();
```

```
// Java program to demonstrate working of ArrayList in Java
import java.io.*;
import java.util.*;

class arrayli
{
    public static void main(String[] args)
                    throws IOException
    {
        // size of ArrayList
        int n = 5;

        //declaring ArrayList with initial size n
        ArrayList<Integer> arrli = new ArrayList<Integer>(n);

        // Appending the new element at the end of the list
        for (int i=1; i<=n; i++)
            arrli.add(i);

        // Printing elements
        System.out.println(arrli);

        // Remove element at index 3
        arrli.remove(3);

        // Displaying ArrayList after deletion
        System.out.println(arrli);

        // Printing elements one by one
        for (int i=0; i<arrli.size(); i++)
            System.out.print(arrli.get(i)+" ");
    }
}
```

Input:

```
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
1 2 3 5
```

Also see – Array vs ArrayList in Java

This article is contributed by **Bharat Sahni**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Technical Scripter
Java-Collections

# PriorityQueue Class in Java

To process the objects in the queue based on the priority, we tend to use Priority Queue.

Important points about Priority Queue:

- PriorityQueue doesn't allow **null**
- We can't create PriorityQueue of Objects that are non-comparable
- The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.
- The head of this queue is the least element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements — ties are broken arbitrarily.
- The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
- It inherits methods from AbstractQueue, AbstractCollection, Collection and Object class.

**Constructor:** PriorityQueue()

This creates a PriorityQueue with the default initial capacity that orders its elements according to their natural ordering.

**Methods:**

1. booleanadd(E element): This method inserts the specified element into this priority queue.
2. public remove(): This method removes a single instance of the specified element from this queue, if it is present
3. public poll(): This method retrieves and removes the head of this queue, or returns null if this queue is empty.
4. public peek(): This method retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.
5. iterator(): Returns an iterator over the elements in this queue.
6. booleancontains(Object o): This method returns true if this queue contains the specified element

Sample code Snippet to show usage of Priority Queue Class:

```
// Java progrm to demonstrate working of priority queue in Java
import java.util.*;

class Example
{
    public static void main(String args[])
    {
        // Creating empty priority queue
        PriorityQueue<String> pQueue =
                      new PriorityQueue<String>();

        // Adding items to the pQueue
        pQueue.add("C");
        pQueue.add("C++");
        pQueue.add("Java");
        pQueue.add("Python");

        // Printing the most priority element
        System.out.println("Head value using peek function:"
                                   + pQueue.peek());

        // Printing all elements
        System.out.println("The queue elements:");
        Iterator itr = pQueue.iterator();
        while (itr.hasNext())
            System.out.println(itr.next());

        // Removing the top priority element (or head) and
        // printing the modified pQueue
        pQueue.poll();
        System.out.println("After removing an element" +
                       "with poll function:");
        Iterator<String> itr2 = pQueue.iterator();
        while (itr2.hasNext())
            System.out.println(itr2.next());

        // Removing Java
        pQueue.remove("Java");
        System.out.println("after removing Java with" +
                       " remove function:");
        Iterator<String> itr3 = pQueue.iterator();
        while (itr3.hasNext())
            System.out.println(itr3.next());

        // Check if an element is present
        boolean b = pQueue.contains("C");
        System.out.println ( "Priority queue contains C" +
                        "ot not?: " + b);

        // get objects from the queue in an array and
        // print the array
        Object[] arr = pQueue.toArray();
        System.out.println ( "Value in array: ");
        for (int i = 0; i<arr.length; i++)
          System.out.println ( "Value: " + arr[i].toString()) ;
    }
}
```

Output:

```
Head value using peek function:C
```

```
The queue elements:
C
C++
Java
Python
After removing an elementwith poll function:
C++
Python
Java
after removing Java with remove function:
C++
Python
Priority queue contains Cot not?: false
Value in array:
Value: C++
Value: Python
```

**Applications :**

Implementing Dijkstra's and Prim's algorithms.

Maximize array sum after K negations

This article is contributed by **Mehak Kumar.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# Deque interface in Java with Example

The java.util.Deque interface is a subtype of the java.util.Queue interface. The Deque is related to the double-ended queue that supports addition or removal of elements from either end of the data structure, it can be used as a queue (first-in-first-out/FIFO) or as a stack (last-in-first-out/LIFO).

**Methods** of deque:

1. **add(element):** Adds an element to the tail.
2. **addFirst(element):** Adds an element to the head.
3. **addLast(element):** Adds an element to the tail.
4. **offer(element):** Adds an element to the tail and returns a boolean to explain if the insertion was successful.
5. **offerFirst(element):** Adds an element to the head and returns a boolean to explain if the insertion was successful.
6. **offerLast(element):** Adds an element to the tail and returns a boolean to explain if the insertion was successful.
7. **iterator():** Returna an iterator for this deque.
8. **descendingIterator():** Returns an iterator that has the reverse order for this deque.
9. **push(element):** Adds an element to the head.
10. **pop(element):** Removes an element from the head and returns it.
11. **removeFirst():** Removes the element at the head.
12. **removeLast():** Removes the element at the tail.

```
// Java program to demonstrate working of
// Deque in Java
import java.util.*;

public class DequeExample
{
    public static void main(String[] args)
    {
        Deque deque = new LinkedList<>();

        // We can add elements to the queue in various ways
        deque.add("Element 1 (Tail)"); // add to tail
        deque.addFirst("Element 2 (Head)");
        deque.addLast("Element 3 (Tail)");
        deque.push("Element 4 (Head)"); //add to head
        deque.offer("Element 5 (Tail)");
        deque.offerFirst("Element 6 (Head)");
        deque.offerLast("Element 7 (Tail)");

        System.out.println(deque + "\n");

        // Iterate through the queue elements.
        System.out.println("Standard Iterator");
        Iterator iterator = deque.iterator();
        while (iterator.hasNext())
            System.out.println("\t" + iterator.next());


        // Reverse order iterator
        Iterator reverse = deque.descendingIterator();
        System.out.println("Reverse Iterator");
        while (reverse.hasNext())
            System.out.println("\t" + reverse.next());

        // Peek returns the head, without deleting
        // it from the deque
        System.out.println("Peek " + deque.peek());
        System.out.println("After peek: " + deque);

        // Pop returns the head, and removes it from
        // the deque
        System.out.println("Pop " + deque.pop());
        System.out.println("After pop: " + deque);

        // We can check if a specific element exists
        // in the deque
        System.out.println("Contains element 3: " +
                        deque.contains("Element 3 (Tail)"));

        // We can remove the first / last element.
        deque.removeFirst();
        deque.removeLast();
        System.out.println("Deque after removing " +
                        "first and last: " + deque);

    }
}
```

Output:

```
[Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail),
 Element 5 (Tail), Element 7 (Tail)]

Standard Iterator
 Element 6 (Head)
 Element 4 (Head)
 Element 2 (Head)
 Element 1 (Tail)
 Element 3 (Tail)
 Element 5 (Tail)
 Element 7 (Tail)
Reverse Iterator
 Element 7 (Tail)
 Element 5 (Tail)
 Element 3 (Tail)
 Element 1 (Tail)
 Element 2 (Head)
 Element 4 (Head)
 Element 6 (Head)
Peek Element 6 (Head)
After peek: [Element 6 (Head), Element 4 (Head), Element 2 (Head), Element 1 (Tail),
 Element 3 (Tail), Element 5 (Tail), Element 7 (Tail)]
Pop Element 6 (Head)
After pop: [Element 4 (Head), Element 2 (Head), Element 1 (Tail), Element 3 (Tail),
 Element 5 (Tail), Element 7 (Tail)]
Contains element 3: true
Deque after removing first and last: [Element 2 (Head), Element 1 (Tail), Element 3 (Tail),
 Element 5 (Tail)]
```

**Deque in Collection Hierarchy**

collectionjava

Image Source : https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# NavigableMap Interface in Java with Example

NavigableMap is an extension of SortedMap which provides convenient navigation method like lowerKey, floorKey, ceilingKey and higherKey, and along with these popular navigation method it also provide ways to create a Sub Map from existing Map in Java e.g. headMap whose keys are less than specified key, tailMap whose keys are greater than specified key and a subMap which is strictly contains keys which falls between toKey and fromKey.

An example class that implements NavigableMap is TreeMap.

**Methods** of NavigableMap:

1. **lowerKey(Object key)** : Returns the greatest key strictly less than the given key, or if there is no such key.
2. **floorKey(Object key)** : Returns the greatest key less than or equal to the given key, or if there is no such key.
3. **ceilingKey(Object key)** : Returns the least key greater than or equal to the given key, or if there is no such key.
4. **higherKey(Object key)** : Returns the least key strictly greater than the given key, or if there is no such key.
5. **descendingMap()** : Returns a reverse order view of the mappings contained in this map.
6. **headMap(object toKey, boolean inclusive)** : Returns a view of the portion of this map whose keys are less than (or equal to, if inclusive is true) toKey.
7. **subMap(object fromKey, boolean fromInclusive, object toKey, boolean toInclusive)** : Returns a view of the portion of this map whose keys range from fromKey to toKey.
8. **tailMap(object fromKey, boolean inclusive)** : Returns a view of the portion of this map whose keys are greater than (or equal to, if inclusive is true) fromKey.

```
// Java program to demonstrate NavigableMap
import java.util.NavigableMap;
import java.util.TreeMap;

public class Example
{
    public static void main(String[] args)
    {
        NavigableMap<String, Integer> nm =
                          new TreeMap<String, Integer>();
        nm.put("C", 888);
        nm.put("Y", 999);
        nm.put("A", 444);
        nm.put("T", 555);
        nm.put("B", 666);
        nm.put("A", 555);
        System.out.printf("Descending Set  : %s%n",
                          nm.descendingKeySet());
        System.out.printf("Floor Entry  : %s%n",
                          nm.floorEntry("L"));
        System.out.printf("First Entry  : %s%n",
                          nm.firstEntry());
        System.out.printf("Last Key : %s%n",
                          nm.lastKey());
        System.out.printf("First Key : %s%n",
                          nm.firstKey());
        System.out.printf("Original Map : %s%n", nm);
        System.out.printf("Reverse Map : %s%n",
                           nm.descendingMap());
    }
}
```

Output:

```
Descending Set : [Y, T, C, B, A]
Floor Entry : C=888
First Entry : A=555
Last Key : Y
First Key : A
Original Map : {A=555, B=666, C=888, T=555, Y=999}
Reverse Map : {Y=999, T=555, C=888, B=666, A=555}
```

collectionjava

Image Source : https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# NavigableSet in Java with Examples

NavigableSet represents a navigable set in Java Collection Framework. The NavigableSet interface inherits from the SortedSet interface. It behaves like a SortedSet with the exception that we have navigation methods available in addition to the sorting mechanisms of the SortedSet. For example, NavigableSet interface can navigate the set in reverse order compared to the order defined in SortedSet.

The classes that implement this interface are, TreeSet and ConcurrentSkipListSet

**Methods** of NavigableSet (Not in SortedSet):

1. Lower(E e) : Returns the greatest element in this set which is less than the given element or NULL if there is no such element.
2. Floor(E e ) : Returns the greatest element in this set which is less than or equal to given element or NULL if there is no such element.
3. Ceiling(E e) : Returns the least element in this set which is greater than or equal to given element or NULL if there is no such element.
4. Higher(E e) : Returns the least element in this set which is greater than the given element or NULL if there is no such element.
5. pollFirst() : Retrieve and remove the first least element. Or return null if there is no such element.
6. pollLast() : Retrieve and remove the last highest element. Or return null if there is no such element.

```
// A Java program to demonstrate working of SortedSet
import java.util.NavigableSet;
import java.util.TreeSet;

public class hashset
{
    public static void main(String[] args)
    {
        NavigableSet<Integer> ns = new TreeSet<>();
        ns.add(0);
        ns.add(1);
        ns.add(2);
        ns.add(3);
        ns.add(4);
        ns.add(5);
        ns.add(6);

        // Get a reverse view of the navigable set
        NavigableSet<Integer> reverseNs = ns.descendingSet();

        // Print the normal and reverse views
        System.out.println("Normal order: " + ns);
        System.out.println("Reverse order: " + reverseNs);

        NavigableSet<Integer> threeOrMore = ns.tailSet(3, true);
        System.out.println("3 or  more:  " + threeOrMore);
        System.out.println("lower(3): " + ns.lower(3));
        System.out.println("floor(3): " + ns.floor(3));
        System.out.println("higher(3): " + ns.higher(3));
        System.out.println("ceiling(3): " + ns.ceiling(3));
        System.out.println("pollFirst(): " + ns.pollFirst());
        System.out.println("Navigable Set:  " + ns);

        System.out.println("pollLast(): " + ns.pollLast());
        System.out.println("Navigable Set:  " + ns);

        System.out.println("pollFirst(): " + ns.pollFirst());
        System.out.println("Navigable Set:  " + ns);

        System.out.println("pollFirst(): " + ns.pollFirst());
        System.out.println("Navigable Set:  " + ns);

        System.out.println("pollFirst(): " + ns.pollFirst());
        System.out.println("Navigable Set:  " + ns);

        System.out.println("pollFirst(): " + ns.pollFirst());
        System.out.println("pollLast(): " + ns.pollLast());
    }
}
```

Output:

```
Normal order: [0, 1, 2, 3, 4, 5, 6]
Reverse order: [6, 5, 4, 3, 2, 1, 0]
3 or  more:  [3, 4, 5, 6]
lower(3): 2
floor(3): 3
higher(3): 4
ceiling(3): 3
pollFirst(): 0
Navigable Set:  [1, 2, 3, 4, 5, 6]
pollLast(): 6
```

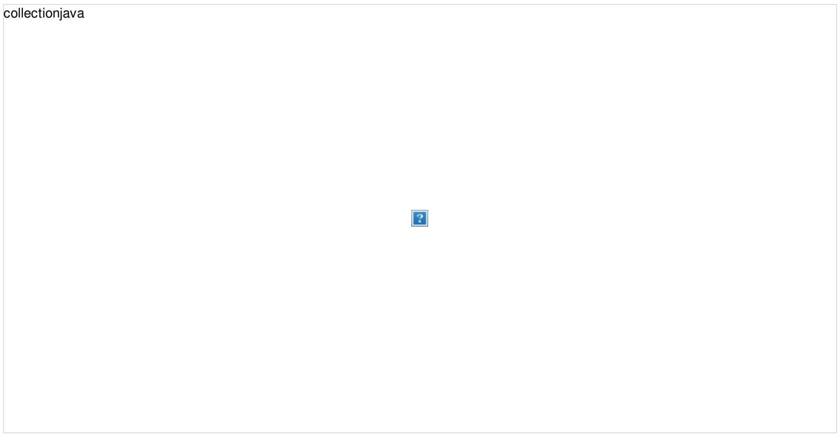

collectionjava

Image Source : https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

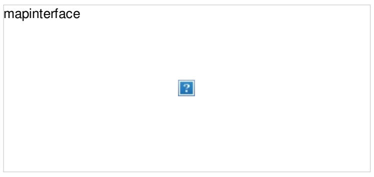
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Map interface in Java with examples

The java.util.Map interface represents a mapping between a key and a value. The Map interface is not a subtype of the Collection interface. Therefore it behaves a bit different from the rest of the collection types.


mapinterface

A Map cannot contain duplicate keys and each key can map to at most one value. Some implementations allow null key and null value (HashMap and LinkedHashMap) but some do not (TreeMap).

The order of a map depends on specific implementations, e.g TreeMap and LinkedHashMap have predictable order, while HashMap does not.
Exampled class that implements this interface is HashMap, TreeMap and LinkedHashMap.

**Why** and **When** Use Maps:
Maps are perfectly for key-value association mapping such as dictionaries. Use Maps when you want to retrieve and update elements by keys, or perform lookups by keys. Some examples:

- A map of error codes and their descriptions.
- A map of zip codes and cities.
- A map of managers and employees. Each manager (key) is associated with a list of employees (value) he manages.
- A map of classes and students. Each class (key) is associated with a list of students (value).

**Methods** of Map:

1. public Object put(Object key, Object value) :- is used to insert an entry in this map.
2. public void putAll(Map map) :- is used to insert the specified map in this map.
3. public Object remove(Object key) :- is used to delete an entry for the specified key.
4. public Object get(Object key) :- is used to return the value for the specified key.
5. public boolean containsKey(Object key) :- is used to search the specified key from this map.
6. public Set keySet() :- returns the Set view containing all the keys.
7. public Set entrySet() :- returns the Set view containing all the keys and values.

```
// Java program to demonstrate working of Map interface
import java.util.*;
class HashMapDemo
{
   public static void main(String args[])
   {
      HashMap< String,Integer> hm =
                     new HashMap< String,Integer>();
      hm.put("a", new Integer(100));
      hm.put("b", new Integer(200));
      hm.put("c", new Integer(300));
      hm.put("d", new Integer(400));

      // Returns Set view
      Set< Map.Entry< String,Integer> > st = hm.entrySet();

      for (Map.Entry< String,Integer> me:st)
      {
         System.out.print(me.getKey()+":");
         System.out.println(me.getValue());
      }
   }
}
```

Output:

```
a:100
b:200
c:300
d:400
```

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

---

## SortedMap Interface in Java with Examples

SortedMap is an interface in collection framework. This interface extends Map inrerface and provides a total ordering of its elements (elements can be traversed in sorted order of keys). Exampled class that implements this interface is TreeMap.

sortedmap

The main characteristic of a SortedMap is that, it orders the keys by their natural ordering, or by a specified comparator. So consider using a TreeMap when you want a map that satisfies the following criteria:

- null key or null value are not permitted.
- The keys are sorted either by natural ordering or by a specified comparator.

**Methods** of SortedMap:

1. subMap(K fromKey, K toKey): Returns a view of the portion of this Map whose keys range from fromKey, inclusive, to toKey, exclusive.
2. headMap(K toKey): Returns a view of the portion of this Map whose keys are strictly less than toKey.
3. tailMap(K fromKey): Returns a view of the portion of this Map whose keys are greater than or equal to fromKey.
4. firstKey(): Returns the first (lowest) key currently in this Map.
5. lastKey(): Returns the last (highest) key currently in this Map.
6. comparator(): Returns the Comparator used to order the keys in this Map, or null if this Map uses the natural ordering of its keys.

**Code** for SortedMap:

```
public interface SortedMap extends Map
{
    Comparator comparator();
    SortedMap subMap(K fromKey, K toKey);
    SortedMap headMap(K toKey);
    SortedMap tailMap(K fromKey);
    K firstKey();
    K lastKey();
}
```

```
// Java code to demonstrate SortedMap
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapExample
{
    public static void main(String[] args)
    {
        SortedMap<Integer, String> sm =
                  new TreeMap<Integer, String>();
        sm.put(new Integer(2), "practice");
        sm.put(new Integer(3), "quiz");
        sm.put(new Integer(5), "code");
        sm.put(new Integer(4), "contribute");
        sm.put(new Integer(1), "geeksforgeeks");
        Set s = sm.entrySet();

        // Using iterator in SortedMap
        Iterator i = s.iterator();

        // Traversing map. Note that the traversal
        // produced sorted (by keys) output .
        while (i.hasNext())
        {
            Map.Entry m = (Map.Entry)i.next();

            int key = (Integer)m.getKey();
            String value = (String)m.getValue();

            System.out.println("Key : " + key +
                            "  value : " + value);
        }
    }
}
```

Output:

```
Key : 1  value : geeksforgeeks
Key : 2  value : practice
Key : 3  value : quiz
Key : 4  value : contribute
Key : 5  value : code
```

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# SortedSet Interface in Java with Examples

SortedSet is an interface in collection framework. This interface extends Set and provides a total ordering of its elements. Exampled class that implements this interface is TreeSet.

sortedsetjava

All elements of a SortedSet must implement the Comparable interface (or be accepted by the specified Comparator) and all such elements must be mutually comparable (i.e, Mutually Comparable simply means that two objects accept each other as the argument to their compareTo method)

**Methods** of Sorted Set interface:

1. **comparator() :** Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements.
2. **first() :** Returns the first (lowest) element currently in this set.
3. **headSet(E toElement) :** Returns a view of the portion of this set whose elements are strictly less than toElement.
4. **last() :** Returns the last (highest) element currently in this set.
5. **subSet(E fromElement, E toElement) :** Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
6. **tailSet(E fromElement) :** Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

```
public interface SortedSet extends Set
{
    // Range views
    SortedSet subSet(E fromElement, E toElement);
    SortedSet headSet(E toElement);
    SortedSet tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator comparator();
}
```

```
// A Java program to demonstrate working of SortedSet
import java.util.SortedSet;
import java.util.TreeSet;

public class Main
{
    public static void main(String[] args)
    {
        // Create a TreeSet and inserting elements
        SortedSet<String> sites = new TreeSet<>();
        sites.add("practice");
        sites.add("geeksforgeeks");
        sites.add("quiz");
        sites.add("code");

        System.out.println("Sorted Set: " + sites);
        System.out.println("First: " + sites.first());
        System.out.println("Last: " + sites.last());

        // Getting elements before quiz (Excluding) in a sortedSet
        SortedSet<String> beforeQuiz = sites.headSet("quiz");
        System.out.println(beforeQuiz);

        // Getting elements between code (Including) and
        // practice (Excluding)
        SortedSet<String> betweenCodeAndQuiz =
                                  sites.subSet("code","practice");
        System.out.println(betweenCodeAndQuiz);

        // Getting elements after code (Including)
        SortedSet<String> afterCode = sites.tailSet("code");
        System.out.println(afterCode);
    }
}
```

Output:

```
Sorted Set:
First: code
Last: quiz
[code, geeksforgeeks, practice]
[code, geeksforgeeks]
[code, geeksforgeeks, practice, quiz]
```

**Reference:**

http://docs.oracle.com/javase/tutorial/collections/interfaces/sorted-set.html

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# How to remove an element from ArrayList in Java?

There are two way to remove an element from ArrayList.

**1. By using remove() methods :**

ArrayList provides two overloaded remove() method.
a. **remove(int index)** : Accept index of object to be removed.
b. **remove(Obejct obj)** : Accept object to be removed.
What happens when we have an **integer** arrayList and we want to remove an item? For example consider below program.

```
// Java program to demonstrate working of remove
// on an integer arraylist
import java.util.List;
import java.util.ArrayList;

public class GFG
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(1);
        al.add(2);

        // This makes a call to remove(int) and
        // removes element 20.
        al.remove(1);

        // Now element 30 is moved one position back
        // So element 30 is removed this time
        al.remove(1);

        System.out.println("Modified ArrayList : " + al);
    }
}
```

Output :

```
Modified ArrayList : [10, 1, 2]
```

We can see that the passed parameter is considered as index. How to remove elements by value.

```
// Java program to demonstrate working of remove
// on an integer arraylist
import java.util.List;
import java.util.ArrayList;

public class GFG
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(1);
        al.add(2);

        // This makes a call to remove(Object) and
        // removes element 1
        al.remove(new Integer(1));

        // This makes a call to remove(Object) and
        // removes element 2
        al.remove(new Integer(2));

        System.out.println("Modified ArrayList : " + al);
    }
}
```

Output :

```
Modified ArrayList : [10, 20, 30]
```

**2. Using Iterator.remove() method :**

It is not recommended to use ArrayList.remove() when iterating over elements. This may lead to ConcurrentModificationException (Refer this for a sample program with this exception). When iterating over elements, it is recommended to use Iterator.remove() method .

```
// Java program to demonstrate working of
// Iterator.remove() on an integer arraylist
```

```
import java.util.List;
import java.util.ArrayList;
import java.util.Iterator;

public class GFG
{
    public static void main(String[] args)
    {
        List al = new ArrayList();
        al.add(10);
        al.add(20);
        al.add(30);
        al.add(1);
        al.add(2);

        // Remove elements smaller than 10 using
        // Iterator.remove()
        Iterator itr = al.iterator();
        while (itr.hasNext())
        {
            int x = (Integer)itr.next();
            if (x < 10)
                itr.remove();
        }

        System.out.println("Modified ArrayList : "
                                    + al);
    }
}
```

Output :

```
Modified ArrayList : [10, 20, 30]
```

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# LinkedHashSet class in Java with Examples

A LinkedHashSet is an ordered version of HashSet that maintains a doubly-linked List across all elements. When the iteration order is needed to be maintained this class in used. When iterating through a HashSet the order is unpredictable, while a LinkedHashSet lets us iterate through the elements in the order in which they were inserted.when cycling through LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

**Syntax:**

```
LinkedHashSet<String> hs = new LinkedHashSet<String>();
```

- Contains unique elements only like HashSet. It extends HashSet class and implements Set interface.
- Maintains insertion order.

Basic **Operations** of LinkedHashSet:

```
import java.util.LinkedHashSet;
public class Demo
{
    public static void main(String[] args)
    {
        LinkedHashSet<String> linkedset =
                        new LinkedHashSet<String>();

        // Adding element to LinkedHashSet
        linkedset.add("A");
        linkedset.add("B");
        linkedset.add("C");
        linkedset.add("D");

        //This will not add new element as A already exists
        linkedset.add("A");
        linkedset.add("E");

        System.out.println("Size of LinkedHashSet = " +
                              linkedset.size());
        System.out.println("Original LinkedHashSet:" + linkedset);
        System.out.println("Removing D from LinkedHashSet: " +
                        linkedset.remove("D"));
        System.out.println("Trying to Remove Z which is not "+
                        "present: " + linkedset.remove("Z"));
        System.out.println("Checking if A is present=" +
                        linkedset.contains("A"));
        System.out.println("Updated LinkedHashSet: " + linkedset);
    }
}
```

**Output:**

```
Size of LinkedHashSet=5
Original LinkedHashSet:[A, B, C, D, E]
Removing D from LinkedHashSet: true
Trying to Remove Z which is not present: false
Checking if A is present=true
Updated LinkedHashSet: [A, B, C, E]
```

LinkedHashmap vs LinkedHashset

- LinkedHashMap does a mapping of keys to values whereas a LinkedHashSet simply stores a collection of things with no duplicates.
- LinkedHashMap extends HashMap and LinkedHashSet extends HashSet.

**Important :** Keeping the insertion order in both LinkedHashmap and LinkedHashset have additional associated costs, both in terms of spending additional CPU cycles and needing more memory. If you do not need the insertion order maintained, it is recommended to use the lighter-weight HashSet and HashMap instead.

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# EnumSet class in Java with Example

EnumSet is one of the specialized implementation of Set interface for an enumeration type. It extends AbstractSet and implements Set Interface in Java.

It is a generic class declared as:

```
public abstract class EnumSet<E extends Enum<E>>
```

Here, E specifies the elements. E must extend Enum, which enforces the requirement that the elements must be of specified enum type.

Reference to full list of methods of this class:https://docs.oracle.com/javase/7/docs/api/java/util/EnumSet.html

**Important:**

- EnumSet class is a member of the Java Collections Framework & is not synchronized.
- It's a high performance set implementation, much faster than HashSet.
- All elements of each EnumSet instance must be elements of a single enum type.

```
// Java program to illustrate working of EnumSet and
// its functions.
import java.util.EnumSet;

enum Gfg
{
    CODE, LEARN, CONTRIBUTE, QUIZ, MCQ
};
public class Example
{
    public static void main(String[] args)
    {
        // create a set
        EnumSet<Gfg> set1, set2, set3, set4;

        // add elements
        set1 = EnumSet.of(Gfg.QUIZ, Gfg.CONTRIBUTE, Gfg.LEARN, Gfg.CODE);
        set2 = EnumSet.complementOf(set1);
        set3 = EnumSet.allOf(Gfg.class);
        set4 = EnumSet.range(Gfg.CODE, Gfg.CONTRIBUTE);
        System.out.println("Set 1: " + set1);
        System.out.println("Set 2: " + set2);
        System.out.println("Set 3: " + set3);
        System.out.println("Set 4: " + set4);
    }
}
```

Output:

```
Set 1: [CODE, LEARN, CONTRIBUTE, QUIZ]
Set 2: [MCQ]
Set 3: [CODE, LEARN, CONTRIBUTE, QUIZ, MCQ]
Set 4: [CODE, LEARN, CONTRIBUTE]
```

**Methods** used in the above example:

- of(E e1, E e2) : Creates an enum set initially containing the specified elements.
- complementOf(EnumSet s) : Creates an enum set with the same element type as the specified enum set, initially containing all the elements of this type that are not contained in the specified set.
- allOf(Class elementType) : Creates an enum set containing all of the elements in the specified element type.
- range(E from, E to) : Creates an enum set initially containing all of the elements in the range defined by the two specified endpoints.

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections
Java-Library

# LinkedHashMap class in Java with Example

HashMap in Java provides quick insert, search and delete operations. However it does not maintain any order on elements inserted into it. If we want to keep track of order of insertion, we can use LinkedHashMap. LinkedHashMap is like HashMap with additional feature that we can access elements in their insertion order.

**Syntax**

```
LinkedHashMap<Integer, String> lhm = new LinkedHashMap<Integer, String>();
```

- A LinkedHashMap contains values based on the key. It implements the Map interface and extends HashMap class.
- It contains only unique elements (See this for details)..
- It may have one null key and multiple null values (See this for details).
- It is same as HashMap with additional feature that it maintains insertion order. For example, when we ran the code with HashMap, we got different oder of elements (See this).

Basic **Operations** of LinkedHashMap class:

```
// Java program to demonstrate working of LinkedHashMap
import java.util.*;

public class BasicLinkedHashMap
{
    public static void main(String a[])
    {
        LinkedHashMap<String, String> lhm =
                       new LinkedHashMap<String, String>();
        lhm.put("one", "practice.geeksforgeeks.org");
        lhm.put("two", "code.geeksforgeeks.org");
        lhm.put("four", "quiz.geeksforgeeks.org");

        // It prints the elements in same order as they were inserted
        System.out.println(lhm);

        System.out.println("Getting value for key 'one': " + lhm.get("one"));
        System.out.println("Size of the map: " + lhm.size());
        System.out.println("Is map empty? " + lhm.isEmpty());
        System.out.println("Contains key 'two'? "+ lhm.containsKey("two"));
        System.out.println("Contains value 'practice.geeksforgeeks.org'? "
                         + lhm.containsValue("practice.geeksforgeeks.org"));
        System.out.println("delete element 'one': " + lhm.remove("one"));
        System.out.println(lhm);
    }
}
```

**Output:**

```
{one=practice.geeksforgeeks.org, two=code.geeksforgeeks.org, four=quiz.geeksforgeeks.org}
Getting value for key 'one': practice.geeksforgeeks.org
Size of the map: 3
Is map empty? false
Contains key 'two'? true
Contains value 'practice.geeksforgeeks.org'? true
delete element 'one': practice.geeksforgeeks.org
{two=code.geeksforgeeks.org, four=quiz.geeksforgeeks.org}
```

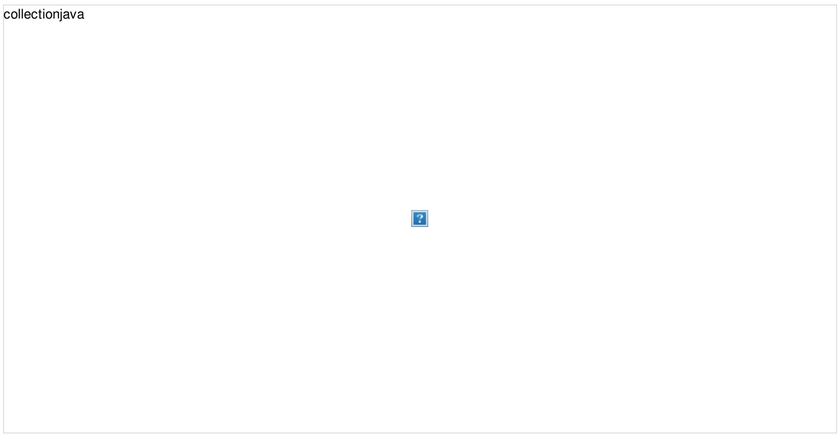


Image Source : https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# TreeSet class in Java with Examples

java.util.TreeSet is implementation class of SortedSet Interface. TreeSet has following important properties.

1. TreeSet implements the SortedSet interface so duplicate values are not allowed.
2. TreeSet does not preserve the insertion order of elements but elements are sorted by keys.
3. TreeSet does not allow to insert Heterogeneous objects. It will throw classCastException at Runtime if trying to add hetrogeneous objects.
4. TreeSet allows to insert null but just once.
5. TreeSet is basically implementation of a self-balancing binary search tree like Red-Black Tree. Therefore operations like add, remove and search take O(Log n) time. And operations like printing n elements in sorted order takes O(n) time.

**Constructors:**

Following are the four constructors in TreeSet class.

1. **TreeSet t = new TreeSet();**
   This will create empty TreeSet object in which elements will get stored in default natural sorting order.
2. **TreeSet t = new TreeSet(Comparator comp);**
   This constructor is used when you externally wants to specify sorting order of elements getting stored.
3. **TreeSet t = new TreeSet(Collection col);**
   This constructor is used when we want to convert any Collection object to TreeSet object.
4. **TreeSet t = new TreeSet(SortedSet s)**;
   This constructor is used to convert SortedSet object to TreeSet Object.

**Synchronized TreeSet:**
Implementation of TreeSet class is not synchronized. If there is need of synchronized version of TreeSet, it can be done externally using Collections.synchronizedSet() method.

```
TreeSet ts = new TreeSet();
Set syncSet = Collections.synchronziedSet(ts);
```

**Adding (or inserting) Elements to TreeSet:**
TreeSet supports add() method to insert elements to it.

```
// Java program to demonstrate insertions in TreeSet
import java.util.*;
```

```
class TreeSetDemo
{
    public static void main (String[] args)
    {
        TreeSet ts1= new TreeSet();
        ts1.add("A");
        ts1.add("B");
        ts1.add("C");

        // Duplicates will not get insert
        ts1.add("C");

        // Elements get stored in default natural
        // Sorting Order(Ascending)
        System.out.println(ts1);  // [A,B,C]

        // ts1.add(2) ; will throw ClassCastException
        //           at run time
    }
}
```

Output :

```
[A, B, C]
```

**Null Insertion:**

If we insert null in non empty TreeSet, it throws NullPointerException because while inserting null it will get compared to existing elements and null can not be compared to any value.

```
// Java program to demonstrate null insertion
// in TreeSet
import java.util.*;

class TreeSetDemo
{
    public static void main (String[] args)
    {
        TreeSet ts2= new TreeSet();
        ts2.add("A");
        ts2.add("B");
        ts2.add("C");
        ts2.add(null); // Throws NullPointerException
    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
 at java.util.TreeMap.put(TreeMap.java:563)
 at java.util.TreeSet.add(TreeSet.java:255)
 at TreeSetDemo.main(File.java:13)
```

We can insert first value as null, but if we insert any more value in TreeSet, it will also throw NullPointerException.

```
// Java program to demonstrate insertion
// in a TreeSet with null
import java.util.*;

class TreeSetDemo
{
    public static void main (String[] args)
    {
        TreeSet ts3 = new TreeSet();
        ts3.add(null);
        ts3.add("A");  // Throws NullPointerException
    }
}
```

Output :

```
Exception in thread "main" java.lang.NullPointerException
 at java.util.TreeMap.compare(TreeMap.java:1294)
 at java.util.TreeMap.put(TreeMap.java:538)
 at java.util.TreeSet.add(TreeSet.java:255)
 at TreeSetDemo.main(File.java:10)
```

**Methods**:

TreeSet implements SortedSet so it has availability of all methods in Collection, Set and SortedSet interfaces. Following are the methods in Treeset interface.

1. **void add(Object o):** This method will add specified element according to some sorting order in TreeSet. Duplicate entires will not get added.

2. **boolean addAll(Collection c):** This method will add all elements of specified Collection to the set. Elements in Collection should be homogeneous otherwise ClassCastException will be thrown.Duplicate Entries of Collection will not be added to TreeSet.

   ```
   // Java program to demonstrate TreeSet creation from
   // ArrayList
   import java.util.*;

   class TreeSetDemo
   {
       public static void main (String[] args)
       {
           ArrayList al = new ArrayList();
           al.add("GeeksforGeeks");
           al.add("GeeksQuiz");
           al.add("Practice");
           al.add("Compiler");
           al.add("Compiler"); //will not be added

           // Creating a TreeSet object from ArrayList
           TreeSet ts4 = new TreeSet(al);

           // [Compiler,GeeksQuiz,GeeksforGeeks,Practice]
           System.out.println(ts4);
       }
   }
   ```

   Output :

   ```
   [Compiler, GeeksQuiz, GeeksforGeeks, Practice]
   ```

3. **void clear() :** This method will remove all the elements.
4. **Comparator comparator():** This method will return Comparator used to sort elements in TreeSet or it will return null if default natural sorting order is used.
5. **boolean contains(Object o):** This method will return true if given element is present in TreeSet else it will return false.
6. **Object first() :** This method will return first element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
7. **Object last():** This method will return last element in TreeSet if TreeSet is not null else it will throw NoSuchElementException.
8. **SortedSet headSet(Object toElement):** This method will return elements of TreeSet which are less than the specified element.
9. **SortedSet tailSet(Object fromElement):** This method will return elements of TreeSet which are greater than or equal to the specified element.
10. **SortedSet subSet(Object fromElement, Object toElement):** This method will return elements ranging from fromElement to toElement. fromElement is inclusive and toElement is exclusive.

```
// Java program to demonstrate TreeSet creation from
// ArrayList
import java.util.*;

class TreeSetDemo
{
    public static void main (String[] args)
    {

        TreeSet ts5 = new TreeSet();

        // Uncommenting below  throws NoSuchElementException
        // System.out.println(ts5.first());

        // Uncommenting below throws NoSuchElementException
        // System.out.println(ts5.last());

        ts5.add("GeeksforGeeks");
        ts5.add("Compiler");
        ts5.add("practice");

        System.out.println(ts5.first()); // Compiler
        System.out.println(ts5.last()); //Practice

        // Elements less than O. It prints
        // [Compiler,GeeksforGeeks]
        System.out.println(ts5.headSet("O"));

        // Elements greater than or equal to G.
        // It prints [GeeksforGeeks, Practice]
        System.out.println(ts5.tailSet("G"));

        // Elements ranging from C to P
        // It prints [Compiler,GeeksforGeeks]
        System.out.println(ts5.subSet("C","P"));

        // Deletes all elements from ts5.
        ts5.clear();

        // Prints nothing
        System.out.println(ts5);
    }
}
```

Output :

```
Compiler
practice
[Compiler, GeeksforGeeks]
[GeeksforGeeks, practice]
[Compiler, GeeksforGeeks]
[]
```

TreeSet in Collection Framework:

Link to the Image: https://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# ArrayList and LinkedList remove() methods in Java with Examples

List interface in Java (which is implemented by ArrayList and LinkedList) provides two versions of remove method.

**remove(Obejct obj) :**

It accepts object to be removed.

- Removes the first occurrence of the specified element from given list, if the element is present. If the element is not present, the given list is not changed.
- After removing, it shifts subsequent elements(if any) to left and decreases their indexes by 1.

It throws following exceptions
ClassCastException – if the type of the specified element is incompatible with this collection
(optional).
NullPointerException – if the specified element is null and this collection does not permit
null elements(optional).
UnsupportedOperationException – if the remove operation is not supported by this collection

```
// A Java program to demonstrate working of list remove
// when Object to be removed is passed.
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        // Demonstrating remove on ArrayList
        List<String>  myAlist = new ArrayList<String>();
        myAlist.add("Geeks");
        myAlist.add("Practice");
        myAlist.add("Quiz");
        System.out.println("Original ArrayList : " + myAlist);
        myAlist.remove("Quiz");
        System.out.println("Modified ArrayList : " + myAlist);

        // Demonstrating remove on LinkedList
        List<String>  myLlist = new ArrayList<String>();
        myLlist.add("Geeks");
        myLlist.add("Practice");
        myLlist.add("Quiz");
        System.out.println("Original LinkedList : " + myLlist);
        myLlist.remove("Quiz");
        System.out.println("Modified LinkedList : " + myLlist);
    }
}
```

Output:

```
Original ArrayList : [Geeks, Practice, Quiz]
Modified ArrayList : [Geeks, Practice]
Original LinkedList : [Geeks, Practice, Quiz]
Modified LinkedList : [Geeks, Practice]
```

**remove(int index) :**

It removes the element at given index

- After removing, it shifts subsequent elements(if any) to left and decreases their indexes by 1.
- If the list contains int types, then this method is called when an int is passed (Please refer this for details)

It throws IndexOutOfBoundsException if index is out of bound,

```
// A Java program to demonstrate working of list remove
// when index is passed.
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        // Demonstrating remove on ArrayList
        List<String>  myAlist = new ArrayList<String>();
        myAlist.add("Geeks");
        myAlist.add("Practice");
        myAlist.add(2);
        System.out.println("Original ArrayList : " + myAlist);
        myAlist.remove("Quiz");
        System.out.println("Modified ArrayList : " + myAlist);

        // Demonstrating remove on LinkedList
        List<String>  myLlist = new ArrayList<String>();
        myLlist.add("Geeks");
        myLlist.add("Practice");
        myLlist.add("Quiz");
        System.out.println("Original LinkedList : " + myLlist);
        myLlist.remove(2);
        System.out.println("Modified LinkedList : " + myLlist);
    }
}
```

Output:

```
Original ArrayList : [Geeks, Practice, Quiz]
Modified ArrayList : [Geeks, Practice]
Original LinkedList : [Geeks, Practice, Quiz]
Modified LinkedList : [Geeks, Practice]
```

**Important Points:**

- Note that there is no direct way to remove elements in array as size of array is fixed. So there are no methods like add(), remove(), delete(). But in Collection like ArrayList and Hashset, we have these methods. So it's better to either convert array to ArrrayList or Use Arraylist from first place when we need these methods.
- It is not recommended to use remove() of list interface when iterating over elements. This may lead to ConcurrentModificationException (Refer this for a sample program with this exception). When iterating over elements, it is recommended to use Iterator.remove() method. Please see this for details.

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# HashMap Class Methods in Java with Examples | Set 1 (put(), get(), isEmpty() and size())

HashMap is a data structure that uses a hash function to map identifying values, known as keys, to their associated values. It contains "key-value" pairs and allows retrieving value by key.

The most impressive feature is it's fast lookup of elements especially for large no. of elements. It is not synchronized by default but we can make it so by calling

```
 Map myhash = Collections.synchronizedMap(hashMap);
```

at creation time, to prevent accidental unsynchronized access to the map.

These are various important hashmap class methods. This post explains: put(), get(), isEmpty() and size()

1. **put():** java.util.HashMap.put() plays role in associating the specified value with the specified key in this map. If the map previously contained a mapping for the key, the old value is replaced.
   **Syntax:**

   ```
   public V put(K key,V value)
   Parameters:
   key - key with which the specified value is to be associated
   ```

```
value - value to be associated with the specified key
Return: the previous value associated with
key, or null if there was no mapping for key.
```

2. **get():** java.util.HashMap.get()method returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
   **Syntax:**

```
public V get(Object key)
Parameters:
key - the key whose associated value is to be returned
Return: the value to which the specified
key is mapped, or null if this map contains no mapping for
the key.
```

3. **isEmpty():** java.util.HashMap.isEmpty() method returns true if the map contains no key-value mappings.
   **Syntax:**

```
public boolean isEmpty()
Return: true if this map contains no key-value mappings
```

4. **size():** java.util.HashMap.size() returns the number of key-value mappings in this map.
   **Syntax:**

```
public int size()
Return: the number of key-value mappings in this map.
```

**Implementation to illustrate above methods**

```
// Java program illustrating use of HashMap methods -
// put(), get(), isEmpty() and size()
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        // Creation of HashMap
        HashMap<String, String> Geeks = new HashMap<>();

        // Adding values to HashMap as ("keys", "values")
        Geeks.put("Language", "Java");
        Geeks.put("Platform", "Geeks For geeks");
        Geeks.put("Code", "HashMap");
        Geeks.put("Learn", "More");

        System.out.println("Testing .isEmpty() method");

        // Checks whether the HashMap is empty or not
        // Not empty so printing the values
        if (!Geeks.isEmpty())
        {
            System.out.println("HashMap Geeks is notempty");

            // Accessing the contents of HashMap through Keys
            System.out.println("GEEKS : " + Geeks.get("Language"));
            System.out.println("GEEKS : " + Geeks.get("Platform"));
            System.out.println("GEEKS : " + Geeks.get("Code"));
            System.out.println("GEEKS : " + Geeks.get("Learn"));

            // size() method prints the size of HashMap.
            System.out.println("Size Of HashMap : " + Geeks.size());
        }
    }
}
```

**Output**

```
Testing .isEmpty() method
HashMap Geeks is notempty
GEEKS : Java
GEEKS : Geeks For geeks
GEEKS : HashMap
GEEKS : More
Size Of HashMap : 4
```

**What are the differences between HashMap and TreeMap?**

1. HashMap implements Map interface while TreeMap implements SortedMap interface. A Sorted Map interface is a child of Map.
2. HashMap implements Hashing, while TreeMap implements Red-Black Tree(a Self Balancing Binary Search Tree). Therefore all differences between Hashing and Balanced Binary Search Tree apply here.
3. Both HashMap and TreeMap have their counterparts HashSet and TreeSet. HashSet and TreeSet implement Set interface. In HashSet and TreeSet, we have only key, no value, these are mainly used to see presence/absence in a set. For above problem, we can't use HashSet (or TreeSet) as we can't store counts. An example problem where we would prefer HashSet (or TreeSet) over HashMap (or TreeMap) is to print all distinct elements in an array.

Refer HashMap and TreeMap for details.

**What are differences between HashMap and HashTable in Java?**

- HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
- HashMap allows one null key and multiple null values whereas Hashtable doesn't allow any null key or value.
- HashMap is generally preferred over HashTable if thread synchronization is not needed
- HashMap is an advanced version and improvement on the Hashtable. HashMap was created later.

Please Refer Differences between HashMap and HashTable in Java for details.

**What are differences between HashMap and HashSet?**

- HashMap stores key value pairs (for example records of students as value and roll number as key) and HashSet stores only keys (for example a set if integers).
- HashSet internally uses HashMap to store keys

Please refer HashSet in Java for details.

**Hashmap methods in Java with Examples | Set 2 (keySet(), values(), containsKey())**

**Reference:**
https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Collections

# Hashmap methods in Java with Examples | Set 2 (keySet(), values(), containsKey()..)

HashMap Class Methods in Java with Examples | Set 1 (put(), get(), isEmpty() and size())

In this post more methods are discussed.

- **keySet(): java.util.HashMap.keySet()** It returns a Set view of the keys contained in this map. The set is backed by the map, so changes to the map are reflected in the set, and vice-versa.

  **Syntax:**
  public Set keySet()
  **Return:** a set view of the keys contained in this map

- **values(): java.util.HashMap.values()** It returns a Collection view of the values contained in this map. The collection is backed by the map, so changes to the map are reflected in the collection, and vice-versa.

  **Syntax:**
  public Collection values()
  **Return:** a collection view of the values contained in
  this map

- **containsKey(): java.util.HashMap.containsKey()** It returns true if this map maps one or more keys to the specified value.

  **Syntax:**
  public boolean containsValue(Object value)
  **Parameters:**
  value - value whose presence in this map is to be tested
  **Return:** true if this map maps one or more keys to
  the specified value

Implementation:

```
// Java program illustrating usage of HashMap class methods
// keySet(), values(), containsKey()
import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        // 1   Creation of HashMap
        HashMap<String, String> Geeks = new HashMap<>();

        // 2   Adding values to HashMap as ("keys", "values")
        Geeks.put("Language", "Java");
        Geeks.put("Platform", "Geeks For geeks");
        Geeks.put("Code", "HashMap");
        Geeks.put("Learn", "More");

        // 3  containsKey() method is to check the presence
        //    of a particluar key
        // Since 'Code' key present here, the condition is true
        if (Geeks.containsKey("Code"))
            System.out.println("Testing .containsKey : " +
                                           Geeks.get("Code"));

        // 4 keySet() method returns all the keys in HashMap
        Set<String> Geekskeys = Geeks.keySet();
        System.out.println("Initial keys  : " + Geekskeys);


        // 5  values() method return all the values in HashMap
        Collection<String> Geeksvalues = Geeks.values();
        System.out.println("Initial values : " + Geeksvalues);

        // Adding new set of key-value
        Geeks.put("Search", "JavaArticle");

        // Again using .keySet() and .values() methods
        System.out.println("New Keys : " + Geekskeys);
        System.out.println("New Values: " + Geeksvalues);
    }
}
```

Output:

```
Testing .containsKey : HashMap
Initial keys  : [Language, Platform, Learn, Code]
Initial values : [Java, Geeks For geeks, More, HashMap]
New Keys : [Language, Platform, Search, Learn, Code]
New Values: [Java, Geeks For geeks, JavaArticle, More, HashMap]
```

- **.entrySet() : java.util.HashMap.entrySet()** method returns a complete set of keys and values present in the HashMap.

  **Syntax:**
  public Set<Map.Entry> entrySet()
  **Return:**
  complete set of keys and values

- **.getOrDefault : java.util.HashMap.getOrDefault()** method returns a default value if there is no value find using the key we passed as an argument in HashMap. If the value for key if present already in the HashMap, it won't do anything to it.
  It is very nice way to assign values to the keys that are not yet mapped, without interfering with the already present set of keys and values.

  **Syntax:**
  default V getOrDefault(Object key,V defaultValue)
  **Parameters:**
  key - the key whose mapped value we need to return
  defaultValue - the default for the keys present in HashMap
  **Return:**
  mapping the unmapped keys with the default value.

- **.replace() : java.util.HashMap.replace(key, value)** or **java.util.HashMap.replace(key, oldvalue, newvalue)** method is a java.util.HashMap class method.
  1st method accepts set of key and value which will replace the already present value of the key with the the new value passed in the argument. If no such set is present replace() method will do nothing.

Meanwhile 2nd method will only replace the already present set of key-old_value if the key and old_Value are found in the HashMap.

```
Syntax:
replace(k key, v value)
        or
replace(k key, v oldvalue, newvalue)
Parameters:
key      - key in set with the old value.
value    - new value we want to be with the specified key
oldvalue - old value in set with the specified key
newvalue - new value we want to be with the specified key
Return:
True - if the value is replaced
Null - if there is no such set present
```

- **.putIfAbsent java.util.HashMap.putIfAbsent(key, value)** method is being used to insert a new key-value set to the HashMap if the respective set is present. Null value is returned if such key-value set is already present in the HashMap.

```
Syntax:
public V putIfAbsent(key, value)
Parameters:
key      - key with which the specified value is associates.
value    - value to associates with the specified key.
```

```
// Java Program illustrating HashMap class methods().
// entrySet(), getOrDefault(), replace(), putIfAbsent
import java.util.*;
public class NewClass
{
 public static void main(String args[])
 {
  // Creation of HashMap
  HashMap<String, String> Geeks = new HashMap<>();

  // Adding values to HashMap as ("keys", "values")
  Geeks.put("Language", "Java");
  Geeks.put("Code", "HashMap");
  Geeks.put("Learn", "More");

  // .entrySet() returns all the keys with their values present in Hashmap
  Set<Map.Entry<String, String>> mappingGeeks = Geeks.entrySet();
  System.out.println("Set of Keys and Values using entrySet() : "+mappingGeeks);
  System.out.println();

  // Using .getOrDefault to access value
  // Here it is Showing Default value as key - "Code" was already present
  System.out.println("Using .getorDefault : "
                          + Geeks.getOrDefault("Code","javaArticle"));

  // Here it is Showing set value as key - "Search" was not present
  System.out.println("Using .getorDefault : "
                          + Geeks.getOrDefault("Search","javaArticle"));
  System.out.println();

  // .replace() method replacing value of key "Learn"
  Geeks.replace("Learn", "Methods");
  System.out.println("working of .replace()  : "+mappingGeeks);
  System.out.println();

  /* .putIfAbsent() method is placing a new key-value
   as they were not present initially*/
  Geeks.putIfAbsent("cool", "HashMap methods");
  System.out.println("working of .putIfAbsent() : "+mappingGeeks);

  /* .putIfAbsent() method is not doing anything
   as key-value were already present */
  Geeks.putIfAbsent("Code", "With_JAVA");
  System.out.println("working of .putIfAbsent() : "+mappingGeeks);

 }
}
```

**Output:**

```
Set of Keys and Values using entrySet() : [Language=Java, Learn=More, Code=HashMap]

Using .getorDefault : HashMap
Using .getorDefault : javaArticle

working of .replace()  : [Language=Java, Learn=Methods, Code=HashMap]

working of .putIfAbsent() : [Language=Java, cool=HashMap methods, Learn=Methods, Code=HashMap]
working of .putIfAbsent() : [Language=Java, cool=HashMap methods, Learn=Methods, Code=HashMap]
```

- **remove(Object key):** Removes the mapping for this key from this map if present.

```
// Java Program illustrating remove() method using Iterator.

import java.util.*;
public class NewClass
{
    public static void main(String args[])
    {
        //  Creation of HashMap
        HashMap<String, String> Geeks = new HashMap<>();

        //  Adding values to HashMap as ("keys", "values")
        Geeks.put("Language", "Java");
        Geeks.put("Platform", "Geeks For geeks");
        Geeks.put("Code", "HashMap");


        //  .entrySet() returns all the keys with their values present in Hashmap
        Set<Map.Entry<String, String>> mappingGeeks = Geeks.entrySet();
        System.out.println("Set of Keys and Values : "+mappingGeeks);
        System.out.println();

        //  Creating an iterator
        System.out.println("Use of Iterator to remove the sets.");
        Iterator<Map.Entry<String, String>> geeks_iterator = Geeks.entrySet().iterator();
        while(geeks_iterator.hasNext())
        {
            Map.Entry<String, String> entry = geeks_iterator.next();
            //  Removing a set one by one using iterator
            geeks_iterator.remove(); // right way to remove entries from Map,
            // avoids ConcurrentModificationException
            System.out.println("Set of Keys and Values : "+mappingGeeks);

        }
    }
}
```

**Output:**

```
Set of Keys and Values : [Language=Java, Platform=Geeks For geeks, Code=HashMap]

Use of Iterator to remove the sets.
Set of Keys and Values : [Platform=Geeks For geeks, Code=HashMap]
Set of Keys and Values : [Code=HashMap]
Set of Keys and Values : []
```

**Advantage:**

If we use for loop, it get translated to Iterator internally but without using Iterator explicitly we can't remove any entry during Iteration.On doing so, Iterator may throw ConcurrentModificationException. So, we use explicit Iterator and while loop to traverse.

**Reference:**

https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

---

# Vector vs ArrayList in Java

ArrayList and Vectors both implements List interface and both use **array(dynamically resizeable)** as data structure internally very much like using an ordinary array .

**Syntax:**

```
ArrayList<T> al = new ArrayList<T>();
Vector<T> v = new Vector<T>();
```

Major Differences between ArrayList and Vector:

1. **Synchronization :** Vector is **synchronized** that means at a time only one thread can access the code while arrayList is **not synchronized** that means multiple threads can work on arrayList at same time. For example, if one thread is performing add operation, then there can be another thread performing remove operation in multithreading environment.
   If multiple threads access arrayList concurrently then we must synchronize the block of the code which modifies the list either structurally or simple modifies element. Structural modification means addition or deletion of element(s) from the list. Setting the value of an existing element is not a structural modification.
2. **Performance: ArrayList is faster** as it is non-synchronized while vector operations give slow performance as they are synchronized(thread-safe). If one thread works on vector has acquired lock on it which makes other thread will has to wait till lock is released.
3. **Data Growth:** ArrayList and Vector **both grow and shrink dynamically** to maintain optimal use of storage. But the way they resized is different. ArrayList increments 50% of current array size if number of elements exceeds its capacity while vector increments 100% means doubles the current array size.
4. **Traversal:** Vector can use both **Enumeration and Iterator** for traversing over elements of vector while ArrayList can only use **Iterator** for traversing.

*Note: ArrayList is preferable when there is no specific requirement to use vector.*

```
// Java Program to illustrate use of ArrayList
// and Vector in Java
import java.io.*;
import java.util.*;

class GFG
{
    public static void main (String[] args)
    {
        // creating an ArrayList
        ArrayList<String> al = new ArrayList<String>();

        // adding object to arraylist
        al.add("Practice.GeeksforGeeks.org");
        al.add("quiz.GeeksforGeeks.org");
        al.add("code.GeeksforGeeks.org");
        al.add("contribute.GeeksforGeeks.org");

        // traversing elements using Iterator'
        System.out.println("ArrayList elements are:");
        Iterator it = al.iterator();
        while (it.hasNext())
            System.out.println(it.next());

        // creating Vector
        Vector<String> v = new Vector<String>();
        v.addElement("Practice");
        v.addElement("quiz");
        v.addElement("code");

        // traversing elements using Enumeration
        System.out.println("\nVector elements are:");
        Enumeration e = v.elements();
        while (e.hasMoreElements())
            System.out.println(e.nextElement());
    }
}
```

Output:

```
ArrayList elements are:
Practice.GeeksforGeeks.org
quiz.GeeksforGeeks.org
code.GeeksforGeeks.org
contribute.GeeksforGeeks.org

Vector elements are:
Practice
```

```
quiz
code
```

**How to choose between ArrayList and Vector?**

- ArrayList is unsynchronized and not thread safe whereas Vecrors are. Only one thread can call methods on a Vector at a time which is a slight overhead but helpful when safety is a concern. Therefore, in a single threaded case arrayList is an obvious choice but in multithreading vectors can be preferred.
- If we don't know how much data we are going to have, but know the rate at which it grows, Vector has advantage since we can set the increment value in vectors.
- ArrayList is newer and faster. If we don't have any explicit requirements for using any of them – we use ArrayList over vector.

This article is contributed by **Nitsdheerendra**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Collections.shuffle() in Java with Examples

**java.util.Collections.shuffle()** is a java.util.Collections class method.

```
// Shuffles mylist
public static void shuffle(List mylist)
This method throws UnsupportedOperationException if the
given list or its list-iterator does not support
the set operation.
```

```
// Java program to demonstrate working of shuffle()
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        ArrayList<String>  mylist = new ArrayList<String>();
        mylist.add("code");
        mylist.add("quiz");
        mylist.add("geeksforgeeks");
        mylist.add("quiz");
        mylist.add("practice");
        mylist.add("qa");

        System.out.println("Original List : \n" + mylist);

        Collections.shuffle(mylist);

        System.out.println("\nShuffled List : \n" + mylist);
    }
}
```

Output:

```

Original List :
[code, quiz, geeksforgeeks, quiz, practice, qa]

Shuffled List :
[qa, quiz, practice, code, quiz, geeksforgeeks]
```

**shuffle(mylist, rndm)**

It shuffles a given list using the user provided source of randomness.

```
// mylist is the list to be shuffled.
// rndm is source of randomness to shuffle the list.
public static void shuffle(List mylist, Random rndm)

It throws  UnsupportedOperationException if the specified list or
its list-iterator does not support the set operation.
```

```
// Java program to demonstrate working of shuffle()
// with user provided source of randomness.
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        ArrayList<String>  mylist = new ArrayList<String>();
        mylist.add("code");
        mylist.add("quiz");
        mylist.add("geeksforgeeks");
        mylist.add("quiz");
        mylist.add("practice");
        mylist.add("qa");

        System.out.println("Original List : \n" + mylist);

        // Here we use Random() to shuffle given list.
        Collections.shuffle(mylist, new Random());
        System.out.println("\nShuffled List with Random() : \n"
                                             + mylist);

        // Here we use Random(3) to shuffle given list.
        Collections.shuffle(mylist, new Random(3));
        System.out.println("\nShuffled List with Random(3) : \n"
                                             + mylist);

        // Here we use Random(3) to shuffle given list.
        Collections.shuffle(mylist, new Random(5));
        System.out.println("\nShuffled List with Random(5) : \n"
                                             + mylist);
    }
}
```

Output:

```
Original List :
[code, quiz, geeksforgeeks, quiz, practice, qa]

Shuffled List with Random() :
[qa, quiz, geeksforgeeks, code, practice, quiz]

Shuffled List with Random(3) :
[quiz, code, quiz, practice, qa, geeksforgeeks]

Shuffled List with Random(5) :
[code, practice, geeksforgeeks, qa, quiz, quiz]
```

**Important Points:**

- This method works by randomly permuting the list elements
- Runs in linear time. If the provided list does not implement the RandomAccess interface, like LinkedList and is large, it first copies the list into an array, then shuffles the array copy, and finally copies array back into the list. This makes sure that the time remains linear.
- It traverses the list backwards, from the last element up to the second, repeatedly swapping a randomly selected element into the "current position". Elements are randomly selected from the portion of the list that runs from the first element to the current position, inclusive. [Source : javadoc]

**Reference:**
https://developer.microej.com/javadoc/microej_4.0/addons/java/util/Collections.html#shuffle-java.util.List-

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Collections.reverseOrder() in Java with Examples

**java.util.Collections.reverseOrder()** method is a java.util.Collections class method.

```
// Returns a comparator that imposes the reverse of
// the natural ordering on a collection of objects
// that implement the Comparable interface.
// The natural ordering is the ordering imposed by
// the objects' own compareTo method
public static  Comparator reverseOrder()
```

**We can the comparator returned by Collections.reverseOrder() to sort a list in descending order.**

```
// Java program to demonstrate working of Collections.reveseOrder()
// to sort a list in descending order
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of Integers
        ArrayList<Integer> al = new ArrayList<Integer>();
        al.add(30);
        al.add(20);
        al.add(10);
        al.add(40);
        al.add(50);

        /* Collections.sort method is sorting the
        elements of ArrayList in descending order. */
        Collections.sort(al, Collections.reverseOrder());

        // Let us print the sorted list
        System.out.println("List after the use of Collection.reverseOrder()"+
                           " and Collections.sort() :\n" + al);
    }
}
```

Output:

```
List after the use of Collection.reverseOrder() and Collections.sort():
[50, 40, 30, 20, 10]
```

**We can use this method with Arrays.sort() also.**

```
// Java program to demonstrate working of Collections.reveseOrder()
// to sort an array in descending order
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create an array to be sorted in descending order.
        Integer [] arr = {30, 20, 40, 10};

        /* Collections.sort method is sorting the
        elements of arr[] in descending order. */
        Arrays.sort(arr, Collections.reverseOrder());

        // Let us print the sorted array
        System.out.println("Array after the use of Collection.reverseOrder()"+
                           " and Arrays.sort() :\n" + Arrays.toString(arr));
    }
}
```

Output:

```
Array after the use of Collection.reverseOrder() and Arrays.sort() :
[40, 30, 20, 10]
```

**public static Comparator reverseOrder(Comparator c)**

It returns a Comparator that imposes reverse order of a passed Comparator object. We can use this method to sort a list in reverse order of user defined Comparator. For example, in the below program, we have created a reverse of user defined comparator to sort students in descending order of roll numbers.

```
// Java program to demonstrate working of
// reverseOrder(Comparator c) to sort students in descending
// order of roll numbers when there is a user defined comparator
// to do reverse.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                           String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                       " " + this.address;
    }
}

class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        // Sorting a list of students in descending order of
        // roll numbers using a Comparator that is reverse of
        // Sortbyroll()
        Comparator c = Collections.reverseOrder(new Sortbyroll());
        Collections.sort(ar, c);

        System.out.println("\nSorted by rollno");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Output :

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
131 aaaa nyc
121 cccc jaipur
111 bbbb london
```

**References:**

https://docs.oracle.com/javase/7/docs/api/java/util/Collections.html#reverseOrder()

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
.

Java
Java-Collections

# Swapping items of a list in Java : Collections.swap() with Example

**java.util.Collections.swap()** method is a java.util.Collections class method. It swaps elements at the specified positions in given list.

```
// Swaps elements at positions "i" and "j" in myList.
public static void swap(List mylist, int i, int j)
```

```
It throws IndexOutOfBoundsException if either i
or j is out of range.
```

```
// Java program to demonstrate working of Collections.swap
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        // Let us create a list with 4 items
        ArrayList<String>  mylist =
                        new ArrayList<String>();
        mylist.add("code");
        mylist.add("practice");
        mylist.add("quiz");
        mylist.add("geeksforgeeks");

        System.out.println("Original List : \n" + mylist);

        // Swap items at indexes 1 and 2
        Collections.swap(mylist, 1, 2);

        System.out.println("\nAfter swap(mylist, 1, 2) : \n"
                           + mylist);

        // Swap items at indexes 1 and 3
        Collections.swap(mylist, 3, 1);

        System.out.println("\nAfter swap(mylist, 3, 1) : \n"
                           + mylist);
    }
}
```

Output:

```
Original List : Original List :
[code, practice, quiz, geeksforgeeks]

After swap(mylist, 1, 2) :
[code, quiz, practice, geeksforgeeks]

After swap(mylist, 3, 1) :
[code, geeksforgeeks, practice, quiz]
```

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Collections.reverse() in Java with Examples

**java.util.Collections.reverse()** method is a java.util.Collections class method. It reverses the order of elements in a list passed as an argument.

```
// Reverses elements of myList and returns Nothing.
// For example, if list contains {1, 2, 3, 4}, it converts
// list to {4, 3, 2, 1}
public static void reverse(List myList)

It throws UnsupportedOperationException if the specified
list or its list-iterator does not support the set operation.
```

**Reversing an ArrayList or LinkedList**.

```
// Java program to demonstrate working of java.utils.
// Collections.reverse()
import java.util.*;

public class ReverseDemo
{
    public static void main(String[] args)
    {
        // Let us create a list of strings
        List<String>  mylist = new ArrayList<String>();
        mylist.add("code");
        mylist.add("practice");
        mylist.add("quiz");
        mylist.add("geeksforgeeks");

        System.out.println("Original List : " + mylist);

        // Here we are using reverse() method
        // to reverse the element order of mylist
        Collections.reverse(mylist);

        System.out.println("Modified List: " + mylist);
    }
}
```

Output:

```
Original List :
Modified List: [geeksforgeeks, quiz, practice, code]
```

For Linkedlist, we just need to replace ArrayList with LinkedList in “List mylist = new ArrayList();”.

Arrays class in Java doesn’t have reverse method. **We can use Collections.reverse() to reverse an array also.**

```
// Java program to demonstrate reversing of array
// with Collections.reverse()
import java.util.*;
```

```
public class ReverseDemo
{
    public static void main(String[] args)
    {
        // Let us create a list of strings
        Integer arr[] = {10, 20, 30, 40, 50};

        System.out.println("Original Array : " +
                            Arrays.toString(arr));

        // Please refer below post for details of asList()
        // http://www.geeksforgeeks.org/array-class-in-java/
        Collections.reverse(Arrays.asList(arr));

        System.out.println("Modified Array : " +
                            Arrays.toString(arr));
    }
}
```

Output:

```
Original Array : [10, 20, 30, 40, 50]
Modified Array : [50, 40, 30, 20, 10]
```

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Collections
Java-Library
Reverse

# Collections.sort() in Java with Examples

**java.util.Collections.sort()** method is present in java.util.Collections class. It is used to sort the elements present in the specified list of Collection in ascending order.
It works similar to java.util.Arrays.sort() method but it is better then as it can sort the elements of Array as well as linked list, queue and many more present in it.

```
public static void sort(List myList)

myList : A List type object we want to sort.

This method doesn't return anything
```

Example:

```

Let us suppose that our list contains
{"Geeks For Geeks", "Friends", "Dear", "Is", "Superb"}

After using Collection.sort(), we obtain a sorted list as
{"Dear", "Friends", "Geeks For Geeks", "Is", "Superb"}
```

**Sorting an ArrayList in ascending order**

```
// Java program to demonstrate working of Collections.sort()
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al);

        // Let us print the sorted list
        System.out.println("List after the use of" +
                        " Collection.sort() :\n" + al);
    }
}
```

Output:

```
List after the use of Collection.sort() :
[Dear, Friends, Geeks For Geeks, Is, Superb]
```

**Sorting an ArrayList in descending order**

```
// Java program to demonstrate working of Collections.sort()
// to descending order.
import java.util.*;

public class Collectionsorting
{
    public static void main(String[] args)
    {
        // Create a list of strings
        ArrayList<String> al = new ArrayList<String>();
        al.add("Geeks For Geeks");
        al.add("Friends");
        al.add("Dear");
        al.add("Is");
        al.add("Superb");

        /* Collections.sort method is sorting the
        elements of ArrayList in ascending order. */
        Collections.sort(al, Collections.reverseOrder());

        // Let us print the sorted list
        System.out.println("List after the use of" +
                           " Collection.sort() :\n" + al);
    }
}
```

Output:

```
List after the use of Collection.sort() :
[Superb, Is, Geeks For Geeks, Friends, Dear]
```

**Sorting an ArrayList according to user defined criteria.**

We can use Comparator Interface for this purpose.

```
// Java program to demonstrate working of Comparator
// interface and Collections.sort() to sort according
// to user defined criteria.
import java.util.*;
import java.lang.*;
import java.io.*;

// A class to represent a student.
class Student
{
    int rollno;
    String name, address;

    // Constructor
    public Student(int rollno, String name,
                               String address)
    {
        this.rollno = rollno;
        this.name = name;
        this.address = address;
    }

    // Used to print student details in main()
    public String toString()
    {
        return this.rollno + " " + this.name +
                           " " + this.address;
    }
}

class Sortbyroll implements Comparator<Student>
{
    // Used for sorting in ascending order of
    // roll number
    public int compare(Student a, Student b)
    {
        return a.rollno - b.rollno;
    }
}

// Driver class
class Main
{
    public static void main (String[] args)
    {
        ArrayList<Student> ar = new ArrayList<Student>();
        ar.add(new Student(111, "bbbb", "london"));
        ar.add(new Student(131, "aaaa", "nyc"));
        ar.add(new Student(121, "cccc", "jaipur"));

        System.out.println("Unsorted");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));

        Collections.sort(ar, new Sortbyroll());

        System.out.println("\nSorted by rollno");
        for (int i=0; i<ar.size(); i++)
            System.out.println(ar.get(i));
    }
}
```

Output :

```
Unsorted
111 bbbb london
131 aaaa nyc
121 cccc jaipur

Sorted by rollno
111 bbbb london
121 cccc jaipur
131 aaaa nyc
```

**Arrays.sort() vs Collections.sort()**

Arrays.sort works for arrays which can be of primitive data type also. Collections.sort() works for objects Collections like ArrayList, LinkedList, etc.

We can use Collections.sort() to sort an array after creating a ArrayList of given array items.

```
// Using Collections.sort() to sort an array
import java.util.*;
public class Collectionsort
{
    public static void main(String[] args)
    {
        // create an array of string objs
        String domains[] = {"Practice", "Geeks",
                            "Code", "Quiz"};

        // Here we are making a list named as Collist
        List colList =
            new ArrayList(Arrays.asList(domains));
        // Collection.sort() method is used here
        // to sort the list elements.
        Collections.sort(colList);

        // Let us print the sorted list
        System.out.println("List after the use of" +
                          " Collection.sort()  :\n" +
                          colList);
    }
}
```

Output:

```
List after the use of Collection.sort()  :
[Code, Geeks, Practice, Quiz]
```

This article is contributed by **Mohit Gupta**. Article is wished to be useful to the esteemed Geeks.
If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Collections

# Generics in Java

Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

**Generic Class**

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()

Note: In Parameter type we can not use primitives like
      'int','char' or 'double'.
```

```
// A Simple Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T>
{
    // An object of type T is declared
    T obj;
    Test(T obj) {  this.obj = obj;  }  // constructor
    public T getObject()  { return this.obj; }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        // instance of Integer type
        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test <String> sObj =
                          new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

Output:

```
15
GeeksForGeeks
```

We can also pass multiple Type parameters in Generic classes.

```
// A Simple Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }

    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG", 15);

        obj.print();
    }
}
```

Output:

```
GfG
15
```

**Generic Functions:**

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Simple Java program to show working of user defined
// Generic functions

class Test
{
    // A Generic method example
    static <T> void genericDisplay (T element)
    {
        System.out.println(element.getClass().getName() +
                           " = " + element);
    }

    // Driver method
    public static void main(String[] args)
    {
         // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("GeeksForGeeks");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

Output :

```
java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0
```

**Advantages of Generics:**

Programs that uses Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use for any type we want.

.

2. Type Safety : Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

   ```
   // A Simple Java program to demonstrate that NOT using
   // generics can cause run time exceptions
   import java.util.*;

   class Test
   {
       public static void main(String[] args)
       {
           // Creatinga an ArrayList without any type specified
           ArrayList al = new ArrayList();

           al.add("Sachin");
           al.add("Rahul");
           al.add(10); // Compiler allows this

           String s1 = (String)al.get(0);
           String s2 = (String)al.get(1);

           // Causes Runtime Exception
           String s3 = (String)al.get(2);
       }
   }
   ```

   Output :

   ```
   Exception in thread "main" java.lang.ClassCastException:
      java.lang.Integer cannot be cast to java.lang.String
    at Test.main(Test.java:19)
   ```

   **How generics solve this problem?**

   At the time of defining ArrayList, we can specify that this list can take only String objects.

   ```
   // Using generics converts run time exceptions into
   // compile time exception.
   import java.util.*;

   class Test
   {
       public static void main(String[] args)
       {
           // Creating a an ArrayList with String specified
           ArrayList <String> al = new ArrayList<String> ();

           al.add("Sachin");
           al.add("Rahul");

           // Now Compiler doesn't allow this
           al.add(10);
   ```

```
        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
```

Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
          ^
```

.

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

```
// We don't need to typecast individual members of ArrayList
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Typecasting is not needed
        String s1 = al.get(0);
        String s2 = al.get(1);
    }
}
```

.

4. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

**References:**
https://docs.oracle.com/javase/tutorial/java/generics/why.html

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## Wildcards in Java

The question mark (?) is known as the wildcard in generic programming . It represents an unknown type. The wildcard can be used in a variety of situations such as the type of a parameter, field, or local variable; sometimes as a return type. Unlike arrays, different instantiations of a generic type are not compatible with each other, not even explicitly. This incompatibility may be softened by the wildcard if ? is used as an actual type parameter.

**Types of wildcards in Java:**

1. **Upper Bounded Wildcards:** These wildcards can be used when you want to relax the restrictions on a variable. For example, say you want to write a method that works on List < integer >, List < double >, and List < number > , you can do this  using an upper bounded wildcard.
   To declare an upper-bounded wildcard, use the wildcard character ('?'), followed by the extends keyword, followed by its upper bound.

   ```
   public static void add(List<? extends Number> list)
   ```

   **Implementation:**

   ```
   //Java program to demonstrate Upper Bounded Wildcards
   import java.util.Arrays;
   import java.util.List;

   class WildcardDemo
   {
    public static void main(String[] args)
    {

        //Upper Bounded Integer List
     List<Integer> list1= Arrays.asList(4,5,6,7);

     //printing the sum of elements in list
     System.out.println("Total sum is:"+sum(list1));

     //Double list
     List<Double> list2=Arrays.asList(4.1,5.1,6.1);

     //printing the sum of elements in list
     System.out.print("Total sum is:"+sum(list2));
    }

    private static double sum(List<? extends Number> list)
    {
     double sum=0.0;
     for (Number i: list)
     {
      sum+=i.doubleValue();
     }

     return sum;
    }
   }
   ```

   **Output:**

   ```
   Total sum is:22.0
   Total sum is:15.299999999999999
   ```

In the above program, list1 and list2 are objects of the List class. list1 is a collection of Integer and list2 is a collection of Double. Both of them are being passed to method sum which has a wildcard that extends Number. This means that list being passed can be of any field or subclass of that field. Here, Integer and Double are subclasses of class Number.

2. **Lower Bounded Wildcards:** It is expressed using the wildcard character ('?'), followed by the super keyword, followed by its lower bound: <? super A>.

   **Syntax:** Collectiontype <? super A>

   **Implementation:**

```
//Java program to demonstrate Lower Bounded Wildcards
import java.util.Arrays;
import java.util.List;

class WildcardDemo
{
 public static void main(String[] args)
 {
     //Lower Bounded Integer List
  List<Integer> list1= Arrays.asList(4,5,6,7);

  //Integer list object is being passed
  printOnlyIntegerClassorSuperClass(list1);

  //Number list
  List<Number> list2= Arrays.asList(4,5,6,7);

  //Integer list object is being passed
  printOnlyIntegerClassorSuperClass(list2);
 }

 public static void printOnlyIntegerClassorSuperClass(List<? super Integer> list)
 {
     System.out.println(list);
 }
}
```

   **Output:**

```
[4, 5, 6, 7]
[4, 5, 6, 7]
```

   Here arguments can be Integer or superclass of Integer(which is Number). The method printOnlyIntegerClassorSuperClass will only take Integer or its superclass objects. However if we pass list of type Double then we will get compilation error. It is because only the Integer field or its superclass can be passed . Double is not the superclass of Integer.

   Use extend wildcard when you want to get values out of a structure and super wildcard when you put values in a structure. Don't use wildcard when you get and put values in a structure.

   Note: You can specify an upper bound for a wildcard, or you can specify a lower bound, but you cannot specify both.

3. **Unbounded Wildcard:** This wildcard type is specified using the wildcard character (?), for example, List. This is called a list of unknown type. These are useful in the following cases
   - When writing a method which can be employed using functionality provided in Object class.
   - When the code is using methods in the generic class that don't depend on the type parameter

   **Implementation:**

```
//Java program to demonstrate Unbounded wildcard
import java.util.Arrays;
import java.util.List;

class unboundedwildcardemo
{
 public static void main(String[] args)
 {

  //Integer List
  List<Integer> list1= Arrays.asList(1,2,3);

  //Double list
  List<Double> list2=Arrays.asList(1.1,2.2,3.3);

  printlist(list1);

  printlist(list2);
 }

 private static void printlist(List<?> list)
 {

  System.out.println(list);
 }
}
```

   Output:

```
[1, 2, 3]
[1.1, 2.2, 3.3]
```

   This article is contributed by **Nishant Sharma**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

   Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

# Assertions in Java

An assertion allows testing the correctness of any assumptions that have been made in the program.

Assertion is achieved using the **assert** statement in Java. While executing assertion, it is believed to be true. If it fails, JVM throws an error named **AssertionError.** It is mainly used for testing purposes during development.

The **assert** statement is used with a Boolean expression and can be written in two different ways.

**First way :**

```
assert expression;
```

**Second way :**

```
assert expression1 : expression2;
```

**Example of Assertion:-**

```
// Java program to demonstrate syntax of assertion
import java.util.Scanner;

class Test
{
    public static void main( String args[] )
    {
        int value = 15;
        assert value >= 20 : " Underweight";
        System.out.println("value is "+value);
    }
}
```

**Output:**

```
value is 15
```

After enabling assertions
**Output:**

```
Exception in thread "main" java.lang.AssertionError: Underweight
```

**Enabling Assertions**

By default, assertions are disabled. We need to run the code as given. The syntax for enabling assertion statement in Java source code is:

```
java –ea Test
```

Or

```
java –enableassertions Test
```

Here, Test is the file name.

**Disabling Assertions**

The syntax for disabling assertions in java are:

```
java –da Test
```

Or

```
java –disableassertions Test
```

Here, Test is the file name.

**Why to use Assertions**
Wherever a programmer wants to see if his/her assumptions are wrong or not.

- To make sure that an unreachable looking code is actually unreachable.
- To make sure that assumptions written in comments are right.

  ```
  if ((x & 1) == 1)
  {  }
  else // x must be even
  { assert (x % 2 == 0); }
  ```

- To make sure default switch case is not reached.
- To check object’s state.
- In the beginning of the method
- After method invocation.

**Assertion Vs Normal Exception Handling**
Assertions are mainly used to check logically impossible situations. For example, they can be used to check the state a code expects before it starts running or state after it finishes running. Unlike normal exception/error handling, assertions are generally disabled at run-time.

**Where to use Assertions**

- Arguments to private methods. Private arguments are provided by developer’s code only and developer may want to check his/her assumptions about arguments.
- Conditional cases.
- Conditions at the beginning of any method.

**Where not to use Assertions**

- Assertions should not be used to replace error messages
- Assertions should not be used to check arguments in the public methods as they may be provided by user. Error handling should be used to handle errors provided by user.
- Assertions should not be used on command line arguments.

**Related Article :**
Assertions in C/C++

This article is contributed by **Rahul Aggarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Annotations in Java

Annotations are used to provide supplement information about a program.

- Annotations start with ‘@’.
- Annotations do not change action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by compiler. See below code for example.

  ```
  /* Java program to demonstrate that annotations are
  ```

```
   not barely comments (This program throws compiler
   error because we have mentioned override, but not
   overridden, we haver overloaded display) */
class Base
{
    public void display()
    {
        System.out.println("Base display()");
    }
}
class Derived extends Base
{
    @Override
    public void display(int x)
    {
        System.out.println("Derived display(int )");
    }

    public static void main(String args[])
    {
        Derived obj = new Derived();
        obj.display();
    }
}
```

Output :

```
10: error: method does not override or implement
   a method from a supertype
```

If we remove parameter (int x) or we remove @override, the program compiles fine.

## Categories of Annotations

There are 3 categories of Annotations:-

**1. Marker Annotations:**

The only purpose is to mark a declaration. These annotations contain no members and do not consist any data. Thus, its presence as an annotation is sufficient. Since, marker interface contains no members, simply determining whether it is present or absent is sufficient. **@Override** is an example of Marker Annotation.

```
Example: - @TestAnnotation()
```

**2. Single value Annotations:**

These annotations contain only one member and allow a shorthand form of specifying the value of the member. We only need to specify the value for that member when the annotation is applied and don't need to specify the name of the member. However in order to use this shorthand, the name of the member must be **value.**

```
Example: - @TestAnnotation("testing");
```

**3. Full Annotations:**

These annotations consist of multiple data members/ name, value, pairs.

```
Example:- @TestAnnotation(owner="Rahul", value="Class Geeks")
```

## Predefined/ Standard Annotations

Java defines seven built-in annotations.

- Four are imported from java.lang.annotation: **@Retention**, **@Documented**, **@Target**, and **@Inherited**.
- Three are included in java.lang: **@Deprecated, @Override** and **@SuppressWarnings**

### @Deprecated Annotation

- It is a marker annotation. It indicates that a declaration is obsolete and has been replaced by a newer form.
- The Javadoc @deprecated tag should be used when an element has been deprecated.
- @deprecated tag is for documentation and @Deprecated annotation is for runtime reflection.
- @deprecated tag has higher priority than @Deprecated annotation when both are together used.

```
public class DeprecatedTest
{
    @Deprecated
    public void Display()
    {
        System.out.println("Deprecatedtest display()");
    }

    public static void main(String args[])
    {
        DeprecatedTest d1 = new DeprecatedTest();
        d1.Display();
    }
}
```

**Output:**

```
Deprecatedtest display()
```

### @Override Annotation

It is a marker annotation that can be used only on methods. A method annotated with **@Override** must override a method from a superclass. If it doesn't, a compile-time error will result (see this for example). It is used to ensure that a superclass method is actually overridden, and not simply overloaded.

Example:-

```
class Base
{
     public void Display()
     {
         System.out.println("Base display()");
     }

     public static void main(String args[])
     {
         Base t1 = new Derived();
         t1.Display();
     }
}
```

```
class Derived extends Base
{
    @Override
    public void Display()
    {
        System.out.println("Derived display()");
    }
}
```

**Output:**

```
Derived display()
```

**@SuppressWarnings**
It is used to inform the compiler to suppress specified compiler warnings. The warnings to suppress are specified by name, in string form. This type of annotation can be applied to any type of declaration.

Java groups warnings under two categories. They are : **deprecation** and **unchecked.**. Any unchecked warning is generated when a legacy code interfaces with a code that use generics.

```
class DeprecatedTest
{
   @Deprecated
   public void Display()
   {
       System.out.println("Deprecatedtest display()");
   }
}

public class SuppressWarningTest
{
   // If we comment below annotation, program generates
   // warning
   @SuppressWarnings({"checked", "deprecation"})
   public static void main(String args[])
   {
       DeprecatedTest d1 = new DeprecatedTest();
       d1.Display();
   }
}
```

**Output:**

```
Deprecatedtest display()
```

**@Documented Annotations**
It is a marker interface that tells a tool that an annotation is to be documented. Annotations are not included by Javadoc comments. Use of @Documented annotation in the code enables tools like Javadoc to process it and include the annotation type information in the generated document.

**@Target**
It is designed to be used only as an annotation to another annotation. **@Target** takes one argument, which must be constant from the **ElementType** enumeration. This argument specifies the type of declarations to which the annotation can be applied. The constants are shown below along with the type of declaration to which they correspond.

| **Target Constant** | **Annotations Can be Applied To** |
|---|---|
| ANNOTATION_TYPE | Another annotation |
| CONSTRUCTOR | Constructor |
| FIELD | Field |
| LOCAL_VARIABLE | Local variable |
| METHOD | Method |
| PACKAGE | Package |
| PARAMETER | Parameter |
| TYPE | Class, Interface, or enumeration |

We can specify one or more of these values in a **@Target**annotation. To specify multiple values, we must specify them within a braces-delimited list. For example, to specify that an annotation applies only to fields and local variables, you can use this @Target annotation: @Target({ElementType.FIELD, ElementType.LOCAL_VARIABLE}) **@Retention Annotation** It determines where and how long the annotation is retent. The 3 values that the @Retention annotation can have:

- **SOURCE:** Annotations will be retained at the source level and ignored by the compiler.
- **CLASS:** Annotations will be retained at compile time and ignored by the JVM.
- **RUNTIME:** These will be retained at runtime.

**@Inherited**
@Inherited is a marker annotation that can be used only on annotation declaration. It affects only annotations that will be used on class declarations. **@Inherited** causes the annotation for a superclass to be inherited by a subclass. Therefore, when a request for a specific annotation is made to the subclass, if that annotation is not present in the subclass, then its superclass is checked. If that annotation is present in the superclass, and if it is annotated with **@Inherited,** then that annotation will be returned.

## User-defined/ Custom Annotations

User-defined annotations can be used to annotate program elements, i.e. variables, constructors, methods, etc. These annotations can be applied just before declaration of an element (constructor, method, classes, etc).
Syntax of Declaration:-

```
[Access Specifier] @interface<AnnotationName>
{
   DataType <Method Name>() [default value];
}
```

- **AnnotationName** is an identifier.
- Parameter should not be associated with method declarations and **throws** clause should not be used with method declaration.
- Parameters will not have a null value but can have a default value.
- ***default value*** is optional.
- Return type of method should be either primitive, enum, string, class name or array of primitive, enum, string or class name type.

```
package source;
// A Java program to demonstrate user defined annotations
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

// user-defined annotation
@Documented
@Retention(RetentionPolicy.RUNTIME)
@ interface TestAnnotation
{
 String Developer() default "Rahul";
```

```
  String Expirydate();
} // will be retained at runtime

// Driver class that uses @TestAnnotation
public class Test
{
    @TestAnnotation(Developer="Rahul", Expirydate="01-10-2020")
    void fun1()
    {
        System.out.println("Test method 1");
    }

    @TestAnnotation(Developer="Anil", Expirydate="01-10-2021")
    void fun2()
    {
        System.out.println("Test method 2");
    }

    public static void main(String args[])
    {
        System.out.println("Hello");
    }
}
```

Output :

```
Hello
```

This article is contributed by **Rahul Agrawal.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

# Serialization and Deserialization in Java with Example

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

serialize-deserialize-java

The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

To make a Java object serializable we implement the **java.io.Serializable** interface.

The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

```
public final void writeObject(Object obj)
                       throws IOException
```

The ObjectInputStream class contains **readObject()** method for deserializing an object.

```
public final Object readObject()
                  throws IOException,
               ClassNotFoundException
```

```
// Java code for serialization and deserialization of a Java object

import java.io.*;

class Demo implements java.io.Serializable
{
    public int a;
    public String b;

    // Default constructor
    public Demo(int a, String b)
    {
        this.a = a;
        this.b = b;
    }

}

class Test
{
    public static void main(String[] args)
    {
        Demo object = new Demo(1, "geeksforgeeks");
        String filename = "file.ser";

        // Serialization
        try
        {
            //Saving of object in a file
            FileOutputStream file = new FileOutputStream(filename);
            ObjectOutputStream out = new ObjectOutputStream(file);

            // Method for serialization of object
            out.writeObject(object);

            out.close();
            file.close();

            System.out.println("Object has been serialized");

        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }


        Demo object1 = null;

        // Deserialization
        try
        {
            // Reading the object from a file
            FileInputStream file = new FileInputStream(filename);
            ObjectInputStream in = new ObjectInputStream(file);

            // Method for deserialization of object
            object1 = (Demo)in.readObject();

            in.close();
            file.close();

            System.out.println("Object has been deserialized ");
            System.out.println("a = " + object1.a);
            System.out.println("b = " + object1.b);
        }

        catch(IOException ex)
        {
            System.out.println("IOException is caught");
        }

        catch(ClassNotFoundException ex)
        {
            System.out.println("ClassNotFoundException is caught");
        }

    }
}
```

Output :

```
Object has been serialized
Object has been deserialized
a = 1
b = geeksforgeeks
```

This article is contributed by Mehak Narang.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Java

# Lambda Expressions in Java 8

Lambda expressions basically express instances of functional interfaces (An interface with single abstract method is called functional interface. An example is java.lang.Runnable). lambda expressions implement the only abstract function and therefore implement functional interfaces

lambda expressions are added in Java 8 and provide below functionalities.

- Enable to treat functionality as a method argument, or code as data.
- A function that can be created without belonging to any class.
- A lambda expression can be passed around as if it was an object and executed on demand.

```
// Java program to demonstrate lambda expressions
// to implement a user defined functional interface.

// A sample functional interface (An interface with
// single abstract method
interface FuncInterface
{
    // An abstract function
    void abstractFun(int x);

    // A non-abstract (or default) function
    default void normalFun()
    {
       System.out.println("Hello");
    }
}

class Test
{
    public static void main(String args[])
    {
        // lambda expression to implement above
        // functional interface. This interface
        // by default implements abstractFun()
        FuncInterface fobj = (int x)->System.out.println(2*x);

        // This calls above lambda expression and prints 10.
        fobj.abstractFun(5);
    }
}
```

Output:

```
10
```

lambda expression

**Syntax:**

```
 lambda operator -> body
```

where lambda operator can be:

- **Zero parameter:**

  ```
  () -> System.out.println("Zero parameter lambda");
  ```

- **One parameter:**–

  ```
  (p) -> System.out.println("One parameter: " + p);
  ```

  It is not mandatory to use parentheses, if the type of that variable can be inferred from the context

- **Multiple parameters :**

  ```
  (p1, p2) -> System.out.println("Multiple parameters: " + p1 + ", " + p2);
  ```

Please note: Lambda expressions are just like functions and they accept parameters just like functions.

```
// A Java program to demonstrate simple lambda expressions
import java.util.ArrayList;
class Test
{
    public static void main(String args[])
    {
        // Creating an ArrayList with elements
        // {1, 2, 3, 4}
        ArrayList<Integer> arrL = new ArrayList<Integer>();
        arrL.add(1);
        arrL.add(2);
        arrL.add(3);
        arrL.add(4);

        // Using lambda expression to print all elements
        // of arrL
        arrL.forEach(n -> System.out.println(n));

        // Using lambda expression to print even elements
        // of arrL
        arrL.forEach(n -> { if (n%2 == 0) System.out.println(n); });
    }
}
```

Output :

```
1
2
3
4
2
4
```

Note that lambda expressions can only be used to implement functional interfaces. In the above example also, the lambda expression implements Consumer Functional Interface.

A Java program to demonstrate working of lambda expression with two arguments.

```
// Java program to demonstrate working of lambda expressions
public class Test
{
    // operation is implemented using lambda expressions
    interface FuncInter1
    {
        int operation(int a, int b);
    }

    // sayMessage() is implemented using lambda expressions
    // above
    interface FuncInter2
    {
        void sayMessage(String message);
    }

    // Performs FuncInter1's operation on 'a' and 'b'
    private int operate(int a, int b, FuncInter1 fobj)
    {
        return fobj.operation(a, b);
    }

    public static void main(String args[])
    {
        // lambda expression for addition for two parameters
        // data type for x and y is optional.
        // This expression implements 'FuncInter1' interface
        FuncInter1 add = (int x, int y) -> x + y;

        // lambda expression multiplication for two parameters
```

```
        // This expression also implements 'FuncInter1' interface
        FuncInter1 multiply = (int x, int y) -> x * y;

        // Creating an object of Test to call operate using
        // different implementations using lambda Expressions
        Test tobj = new Test();

        // Add two numbers using lambda expression
        System.out.println("Addition is " +
                        tobj.operate(6, 3, add));

        // Multiply two numbers using lambda expression
        System.out.println("Multiplication is " +
                        tobj.operate(6, 3, multiply));

        // lambda expression for single parameter
        // This expression implements 'FuncInter2' interface
        FuncInter2 fobj = message ->System.out.println("Hello "
                                        + message);
        fobj.sayMessage("Geek");
    }
}
```

Output:

```
Addition is 9
Multiplication is 18
Hello Geek
```

**Important points:**

- The body of a lambda expression can contain zero, one or more statements.
- When there is a single statement curly brackets are not mandatory and the return type of the anonymous function is the same as that of the body expression.
- When there are more than one statements, then these must be enclosed in curly brackets (a code block) and the return type of the anonymous function is the same as the type of the value returned within the code block, or void if nothing is returned.

Also see Lambda Expression-QA

This article is contributed by **Sampada Kaushal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Stream In Java

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.
The features of Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-
**Intermediate Operations:**

1. **map:** The map method is used to map the items in the collection to other objects according to the Predicate passed as argument.
   `List number = Arrays.asList(2,3,4,5);`
   `List square = number.stream().map(x->x*x).collect(Collectors.toList());`
2. **filter:** The filter method is used to select elements as per the Predicate passed as argument.
   `List names = Arrays.asList("Reflection","Collection","Stream");`
   `List result=ames.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());`
3. **sorted:** The sorted method is used to sort the stream.
   `List names = Arrays.asList("Reflection","Collection","Stream");`
   `List result = names.stream().sorted().collect(Collectors.toList());`

**Terminal Operations:**

1. **collect:** The collect method is used to return the result of the intermediate operations performed on the stream.
   `List number = Arrays.asList(2,3,4,5,3);`
   `Set square = number.stream().map(x->x*x).collect(Collectors.toSet());`
2. **forEach:** The forEach method is used to iterate through every element of the stream.
   `List number = Arrays.asList(2,3,4,5);`
   `number.stream().map(x->x*x).forEach(y->System.out.println(y));`
3. **reduce:** The reduce method is used to reduce the elements of a stream to a single value.
   The reduce method takes a BinaryOperator as a parameter.
   `List number = Arrays.asList(2,3,4,5);`
   `int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);`

   Here ans variable is assigned 0 as the initial value and i is added to it .

**Program to demonstrate the use of Stream**

```
//a simple program to demonstrate the use of stream in java
import java.util.*;
import java.util.stream.*;

class Demo
{
  public static void main(String args[])
  {

    // create a list of integers
    List<Integer> number = Arrays.asList(2,3,4,5);

    // demonstration of map method
    List<Integer> square = number.stream().map(x -> x*x).
                        collect(Collectors.toList());
    System.out.println(square);
    // create a list of String
    List<String> names =
                Arrays.asList("Reflection","Collection","Stream");

    // demonstration of filter method
    List<String> result = names.stream().filter(s->s.startsWith("S")).
                         collect(Collectors.toList());
    System.out.println(result);

    // demonstration of sorted method
    List<String> show =
            names.stream().sorted().collect(Collectors.toList());
    System.out.println(show);

    // create a list of integers
    List<Integer> numbers = Arrays.asList(2,3,4,5,2);

    // collect method returns a set
    Set<Integer> squareSet =
         numbers.stream().map(x->x*x).collect(Collectors.toSet());
    System.out.println(squareSet);

    // demonstration of forEach method
    number.stream().map(x->x*x).forEach(y->System.out.println(y));

    // demonstration of reduce method
    int even =
       number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

    System.out.println(even);
  }
}
```

Output:

```
[4, 9, 16, 25]
[Stream]
[Collection, Reflection, Stream]
[16, 4, 9, 25]
4
9
16
25
6
```

**Important Points/Observations:**

1. A stream consists of source followed by zero or more intermediate methods combined together (pipelined) and a terminal method to process the objects obtained from the source as per the methods described.
2. Stream is used to compute elements as per the pipelined methods without altering the original value of the object.

This article is contributed by **Akash Ojha** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# BigInteger Class in Java

BigInteger class is used for mathematical operation which involves very big integer calculations that are outside the limit of all available primitive data types.

For example factorial of 100 contains 158 digits in it so we can't store it in any primitive data type available. We can store as large Integer as we want in it. There is no theoretical limit on the upper bound of the range because memory is allocated dynamically but practically as memory is limited you can store a number which has Integer.MAX_VALUE number of bits in it which should be sufficient to store mostly all large values.

Below is an example Java program that uses BigInteger to compute Factorial.

```
// Java program to find large factorials using BigInteger
import java.math.BigInteger;
import java.util.Scanner;

public class Example
{
    // Returns Factorial of N
    static BigInteger factorial(int N)
    {
        // Initialize result
        BigInteger f = new BigInteger("1"); // Or BigInteger.ONE

        // Multiply f with 2, 3, ...N
        for (int i = 2; i <= N; i++)
            f = f.multiply(BigInteger.valueOf(i));

        return f;
    }

    // Driver method
    public static void main(String args[]) throws Exception
    {
        int N = 20;
        System.out.println(factorial(N));
    }
}
```

Output:

```
2432902008176640000
```

If we have to write above program in C++, that would be too large and complex, we can look at Factorail of Large Number.
In this way BigInteger class is very handy to use because of its large method library and it is also used a lot in competitive programming.

Now below is given a list of simple statements in primitive arithmetic and its analogous statement in terms of BigInteger objects.

**Declaration**

```
int a, b;
```

```
BigInteger A, B;
```

**Initialization:**

```
a = 54;
b = 23;
A  = BigInteger.valueOf(54);
B  = BigInteger.valueOf(37);
```

And for Integers available as string you can initialize them as:

```
A  = new BigInteger("54");
B  = new BigInteger("123456789123456789");
```

Some constant are also defined in BigInteger class for ease of initialization :

```
A = BigInteger.ONE;
// Other than this, available constant are BigInteger.ZERO
// and BigInteger.TEN
```

Mathematical operations:

```
int c = a + b;
BigInteger C = A.add(B);
```

Other similar function are subtract() , multiply(), divide(), remainder()

But all these function take BigInteger as their argument so if we want these operation with integers or string convert them to BigInteger before passing them to functions as shown below:

```
String str = "123456789";
BigInteger C = A.add(new BigInteger(str));
int val  = 123456789;
BigInteger C = A.add(BigIntger.valueOf(val));
```

***Extraction of value from BigInteger:***

```
int x   =  A.intValue();   // value should be in limit of int x
long y   = A.longValue();  // value should be in limit of long y
String z = A.toString();
```

***Comparison:***

if (a

Actually  compareTo returns -1(less than), 0(Equal), 1(greater than)  according to values.

For equality we can also use:

```
if (A.equals(B)) {}  // A is equal to B
```

BigInteger class also provides quick methods for prime numbers.

***SPOJ Problems:***

So after above knowledge of function of BigInteger class, we can solve many complex problem easily, but remember as BigInteger class internally uses array of integers for processing, the operation on object of BigIntegers are not as

http://www.spoj.com/problems/JULKA/

http://www.spoj.com/problems/MCONVERT/en/

http://www.spoj.com/problems/MUL2COM/en/

http://www.spoj.com/problems/REC/

http://www.spoj.com/problems/SETNJA/en/

http://www.spoj.com/problems/TREE/

http://www.spoj.com/problems/WORMS/

For more functions and detail refer http://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html

This article is contributed by Utkarsh Trivedi. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner

## Company Wise Coding Practice

Java
large-numbers

# Image Processing in Java | Set 1 (Read and Write)

In this set we will learn how to **read** and **write** image file in Java.

**Classes** required to perform the operation:

1. To read and write image file we have to import the File class [ import java.io.File; ]. This class represents file and directory path names in general.
2. To handle errors we use the IOException class [ import java.io.IOException; ]
3. To hold the image we create the BufferedImage object for that we use BufferedImage class [ import java.awt.image.BufferedImage; ]. This object is used to store an image in RAM.
4. To perform the image read write operation we will import the ImageIO class [ import javax.imageio.ImageIO;]. This class has static methods to read and write an image.

```
// Java program to demonstrate read and write of image
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class MyImage
{
    public static void main(String args[])throws IOException
    {
        int width = 963;    //width of the image
        int height = 640;   //height of the image

        // For storing image in RAM
        BufferedImage image = null;

        // READ IMAGE
        try
        {
            File input_file = new File("G:\\Inp.jpg"); //image file path

            /* create an object of BufferedImage type and pass
               as parameter the width,  height and image int
               type.TYPE_INT_ARGB means that we are representing
               the Alpha, Red, Green and Blue component of the
               image pixel using 8 bit integer value. */
            image = new BufferedImage(width, height,
                                    BufferedImage.TYPE_INT_ARGB);

             // Reading input file
            image = ImageIO.read(input_file);

            System.out.println("Reading complete.");
        }
        catch(IOException e)
        {
            System.out.println("Error: "+e);
        }

        // WRITE IMAGE
        try
        {
            // Output file path
            File output_file = new File("G:\\Out.jpg");

            // Writing to file taking type and path as
            ImageIO.write(image, "jpg", output_file);

            System.out.println("Writing complete.");
        }
        catch(IOException e)
        {
            System.out.println("Error: "+e);
        }
    }//main() ends here
}//class ends here
```

**Note :** This code will not run on online IDE as it needs an image on disk.

In the next set we will be learning how to get and set the pixel value of images in JAVA.

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

# Image Processing In Java | Set 2 (Get and set Pixels)

We strongly recommend to refer below post as a prerequisite of this.
Image Processing in Java | Set 1 (Read and Write)

In this set, we will learn about pixels of image. How we can **get** pixel values of image and how to **set** pixel values in an image using Java programming language.
**Pixels** are the smallest unit of an image which consists of four components Alpha (transparency measure), Red, Green, Blue and in short (ARGB).
The value of all the components lie between 0 and 255 both inclusive. Zero means the component is absent and 255 means the component is fully present.

**Note:**
Since, $2^8$ = 256 and the value of the pixel components lie between 0 and 255, so we need only 8-bits to store the values.
SO, total number of bits required to store the ARGB values is 8*4=32 bits or 4 bytes.
As the order signifies Alpha acquires leftmost 8 bits, Blue acquires rightmost 8 bits.
Thus the bit position :
For blue component being 7-0,
For green component being 15-8,
For red component being 23-16,
For alpha component being 31-24,

Pictorial representation of indexes:

aa

Image source: https://www.safaribooksonline.com/library/view/learning-java-4th/9781449372477/ch21s02.html

```
// Java program to demonstrate get and set pixel
// values of an image
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class GetSetPixels
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        //read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        //get image width and height
        int width = img.getWidth();
        int height = img.getHeight();

        /*  Since, Inp.jpg is a single pixel image so, we
            will not be using the width and height variable */

        /* get pixel value (the arguments in the getRGB method
         denotes the  cordinates of the image from which the
         pixel values need to be extracted) */
        int p = img.getRGB(0,0);

        /* We, have seen that the components of pixel occupy
           8 bits. To get the bits we have to first right shift
           the 32 bits of the pixels by bit position(such as 24
           in case of alpha) and then bitwise ADD it with 0xFF.
           0xFF is the hexadecimal representation of the decimal
           value 255.  */

        // get alpha
        int a = (p>>24) & 0xff;

        // get red
        int r = (p>>16) & 0xff;

        // get green
        int g = (p>>8) & 0xff;

        // get blue
        int a = p & 0xff;

        /*
        for simplicity we will set the ARGB
        value to 255, 100, 150 and 200 respectively.
        */
        a = 255;
        r = 100;
        g = 150;
        b = 200;

        //set the pixel value
        p = (a<<24) | (r<<16) | (g<<8) | b;
        img.setRGB(0, 0, p);

        //write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

**Note** : This code will not run on online IDE as it needs an image on disk.

In the next set we will be learning how to convert a colored image to grayscale image in JAVA.

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner**    **Company Wise Coding Practice**

Java
Image-Processing

# Image Processing in Java | Set 3 (Colored image to greyscale image conversion)

We strongly recommend to refer below post as a prerequisite of this.

- Image Processing in Java | Set 1 (Read and Write)
- Image Processing In Java | Set 2 (Get and set Pixels)

In this set we will be converting a coloured image to greyscale image.

**Note**(Think intuitively): In a greyscale image the Alpha component of the image will be same as the original image, but the RGB wil be changed i.e, all three RGB components will be having a same value for each pixels.

**Algorithm:**

1. Get the RGB value of the pixel.
2. Find the average of RGB i.e., Avg = (R+G+B)/3
3. Replace the R, G and B value of the pixel with average (Avg) calculated in step 2.
4. Repeat Step 1 to Step 3 for each pixels of the image.

**Implementation** of the above algorithm:

```java
// Java program to demonstrate colored to greyscale conversion
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class Grayscale
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        // read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        // get image's width and height
        int width = img.getWidth();
        int height = img.getHeight();

        // convert to greyscale
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                // Here (x,y)denotes the coordinate of image
                // for modifying the pixel value.
                int p = img.getRGB(x,y);

                int a = (p>>24)&0xff;
                int r = (p>>16)&0xff;
                int g = (p>>8)&0xff;
                int b = p&0xff;

                // calculate average
                int avg = (r+g+b)/3;

                // replace RGB value with avg
                p = (a<<24) | (avg<<16) | (avg<<8) | avg;

                img.setRGB(x, y, p);
            }
        }

        // write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

Note : This code will not run on online IDE as it needs an image on disk.

Inp.jpg

Inp

Out.jpg

Out

In the next set we will be learning how to convert a colored image to negative image in JAVA.

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Image Processing in Java | Set 4 (Colored image to Negative image conversion)

We strongly recommend to refer below post as a prerequisite of this.

- Image Processing in Java | Set 1 (Read and Write)
- Image Processing In Java | Set 2 (Get and set Pixels)

In this set we will be converting a colored image to negative image.

**Note**: In a negative image the Alpha component of the image will be same as the original image, but the RGB will be changed i.e, all three RGB components will be having a value of 255-original component value.

**Algorithm:**

1. Get the RGB value of the pixel.
2. Calculate new RGB values as follows:
   - R = 255 – R
   - G = 255 – G
   - B = 255 – B
3. Replace the R, G and B value of the pixel with the values calculated in step 2.
4. Repeat Step 1 to Step 3 for each pixels of the image.

**Implementation** of the above algorithm:

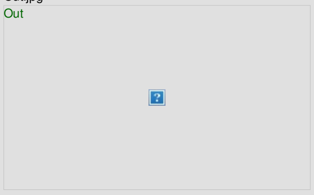
```
// Java program to demonstrate colored to negative conversion
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class Negative
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        // read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        // Get image width and height
        int width = img.getWidth();
        int height = img.getHeight();

        // Convert to negative
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                int p = img.getRGB(x,y);
                int a = (p>>24)&0xff;
                int r = (p>>16)&0xff;
                int g = (p>>8)&0xff;
                int b = p&0xff;

                //subtract RGB from 255
                r = 255 - r;
                g = 255 - g;
                b = 255 - b;

                //set new RGB value
                p = (a<<24) | (r<<16) | (g<<8) | b;
                img.setRGB(x, y, p);
            }
        }

        // write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

Note : This code will not run on online IDE as it needs an image on disk.

Inp.jpg
Inp

Out.jpg

Out



This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Image-Processing

# Image Processing in Java | Set 5 (Colored to Red Green Blue Image Conversion)

We strongly recommend to refer below post as a prerequisite of this.

- Image Processing in Java | Set 1 (Read and Write)
- Image Processing In Java | Set 2 (Get and set Pixels)

In this set we will be converting a colored image to an image with either red effect, green effect or blue effect.

The basic idea is to get the pixel value for each cordinates and then keep the desired resultant color pixel value to be same and set the other two as zero.

**Converting to Red Colored Image**

**Algorithm** for converting an colored image to red colored:

1. Get the RGB value of the pixel.
2. Set the RGB values as follows:
   - R: NO CHANGE
   - G: Set to 0
   - B: Set to 0
3. Replace the R, G and B value of the pixel with the values calculated in step 2.
4. Repeat Step 1 to Step 3 for each pixels of the image.

**Implementation** of the above algorithm:

```java
// Java program to demonstrate colored to red colored image conversion
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class RedImage
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        // read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        // get width and height
        int width = img.getWidth();
        int height = img.getHeight();

        // convert to red image
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                int p = img.getRGB(x,y);

                int a = (p>>24)&0xff;
                int r = (p>>16)&0xff;

                // set new RGB
                // keeping the r value same as in original
                // image and setting g and b as 0.
                p = (a<<24) | (r<<16) | (0<<8) | 0;

                img.setRGB(x, y, p);
            }
        }

        // write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

Note : This code will not run on online IDE as it needs an image on disk.
Output:

```
Inp.jpg
Inp

Oup.jpg
Out red
```

**Converting to Green Colored Image**

**Algotithm** for converting an colored image to green colored:

1. Get the RGB value of the pixel.
2. Set the RGB values as follows:
   - R: Set to 0
   - G: NO CHANGE
   - B: Set to 0
3. Replace the R, G and B value of the pixel with the values calculated in step 2.
4. Repeat Step 1 to Step 3 for each pixels of the image.

**Implementation** of the above algorithm:

```
// Java program to demonstrate colored to green coloured
// image conversion
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class GreenImage
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        //read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        // get width and height
        int width = img.getWidth();
        int height = img.getHeight();

        // convert to green image
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                int p = img.getRGB(x,y);

                int a = (p>>24)&0xff;
                int g = (p>>8)&0xff;

                // set new RGB
                // keeping the g value same as in original
                // image and setting r and b as 0.
                p = (a<<24) | (0<<16) | (g<<8) | 0;

                img.setRGB(x, y, p);
            }
        }

        //write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

Note : This code will not run on online IDE as it needs an image on disk.
Output:

```
Inp.jpg
```

Inp

Oup.jpg
Out green

**Converting to Blue Colored Image**

**Algorithm** for converting an colored image to blue colored:

1. Get the RGB value of the pixel.
2. Set the RGB values as follows:
   - R: Set to 0
   - G: Set to 0
   - B: NO CHANGE
3. Replace the R, G and B value of the pixel with the values calculated in step 2.
4. Repeat Step 1 to Step 3 for each pixels of the image.

**Implementation** of the above algorithm:

```
// Java program to demonstrate colored to blue coloured image conversion
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class BlueImage
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        //read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        // get width and height
        int width = img.getWidth();
        int height = img.getHeight();

        // convert to blue image
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                int p = img.getRGB(x,y);

                int a = (p>>24)&0xff;
                int b = p&0xff;

                // set new RGB
                // keeping the b value same as in original
                // image and setting r and g as 0.
                p = (a<<24) | (0<<16) | (0<<8) | b;

                img.setRGB(x, y, p);
            }
        }

        // write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

Note : This code will not run on online IDE as it needs an image on disk.

Output:

Inp.jpg

Inp

Oup.jpg
Out

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

# Image Procesing in Java | Set 6 (Colored image to Sepia image conversion)

We strongly recommend to refer below post as a prerequisite of this.

- Image Processing in Java | Set 1 (Read and Write)
- Image Processing In Java | Set 2 (Get and set Pixels)

In this set we will be converting a colored image to sepia image.

In a sepia image the Alpha component of the image will be same as the original image(since alpha component denotes the transparency), but the RGB will be changed, which will be calculated by the following formula.

```
newRed = 0.393*R + 0.769*G + 0.189*B
newGreen = 0.349*R + 0.686*G + 0.168*B
newBlue = 0.272*R + 0.534*G + 0.131*B

If any of these output values is greater than 255,
simply set it to 255.
These specific values are the values for sepia tone
that are recommended by Microsoft.
```

**Algorithm:**

1. Get the RGB value of the pixel.
2. Calculate newRed, newGree, newBlue using the above formula (Take the integer value)
3. Set the new RGB value of the pixel as per the following condition:
   - If newRed > 255 then R = 255 else R = newRed
   - If newGreen > 255 then G = 255 else G = newGreen
   - If newBlue > 255 then B = 255 else B = newBlue
4. Replace the value of R, G and B with the new value that we calculated for the pixel.
5. Repeat Step 1 to Step 4 for each pixels of the image.

**Implementation** of the above algorithm:

```
// Java program to demonstrate colored to sepia conversion
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;

public class Sepia
{
    public static void main(String args[])throws IOException
    {
        BufferedImage img = null;
        File f = null;

        //read image
        try
        {
            f = new File("G:\\Inp.jpg");
            img = ImageIO.read(f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }

        // get width and height of the image
        int width = img.getWidth();
        int height = img.getHeight();

        //convert to sepia
        for(int y = 0; y < height; y++)
        {
            for(int x = 0; x < width; x++)
            {
                int p = img.getRGB(x,y);

                int a = (p>>24)&0xff;
                int R = (p>>16)&0xff;
                int G = (p>>8)&0xff;
                int B = p&0xff;

                //calculate newRed, newGreen, newBlue
                int newRed = (int)(0.393*R + 0.769*G + 0.189*B);
                int newGreen = (int)(0.349*R + 0.686*G + 0.168*B);
                int newBlue = (int)(0.272*R + 0.534*G + 0.131*B);

                //check condition
                if (newRed > 255)
                    R = 255;
                else
                    R = newRed;

                if (newGreen > 255)
                    G = 255;
                else
                    G = newGreen;

                if (newBlue > 255)
                    B = 255;
                else
                    B = newBlue;

                //set new RGB value
                p = (a<<24) | (R<<16) | (G<<8) | B;

                img.setRGB(x, y, p);
            }
        }

        //write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println(e);
        }
    }
}
```

**Note** : This code will not run on online IDE as it needs an image on disk.

Output:

```
Inp.jpg
Inp

Out.jpg
Out
```

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

# Image Processing in Java | Set 7 (Creating a random pixel image)

We strongly recommend to refer below post as a prerequisite of this.

- Image Processing in Java | Set 1 (Read and Write)
- Image Processing In Java | Set 2 (Get and set Pixels)

In this set we will be creating a random pixel image. For creating a random pixel image, we don't need any input image We can create an image file and set its pixel values generated randomly.

**Algorithm:**

1. Set the dimension of new image file.
2. Create a BufferedImage object to hold the image [ import java.awt.image.BufferedImage; ]. This object is used to store an image in RAM.
3. Generate random number values for alpha, red, green and blue components.
4. Set the randomly generated ARGB (Alpha, Red, Green and Blue) values.
5. Repeat the steps 3 and 4 for each pixels of the image.

**Implementation** of the above algorithm:

```
// Java program to demonstrate creation of random pixel image
import java.io.File;
import java.io.IOException;
import java.awt.image.BufferedImage;
import javax.imageio.ImageIO;
public class RandomImage
```

```
{
    public static void main(String args[])throws IOException
    {
        // Image file dimensions
        int width = 640, height = 320;

        // Create buffered image object
        BufferedImage img = null;
        img = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);

        // file object
        File f = null;

        // create random values pixel by pixel
        for (int y = 0; y < height; y++)
        {
            for (int x = 0; x < width; x++)
            {
                int a = (int)(Math.random()*256); //generating
                int r = (int)(Math.random()*256); //values
                int g = (int)(Math.random()*256); //less than
                int b = (int)(Math.random()*256); //256

                int p = (a<<24) | (r<<16) | (g<<8) | b; //pixel

                img.setRGB(x, y, p);
            }
        }

        // write image
        try
        {
            f = new File("G:\\Out.jpg");
            ImageIO.write(img, "jpg", f);
        }
        catch(IOException e)
        {
            System.out.println("Error: " + e);
        }
    }
}
```

**Note :** Code will not run on online ide since it writes image in drive.

Output:

```
Out.jpg
Out
```

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

---

# Image Processing in Java | Set 9 ( Face Detection )

In the introductory set on Image Processing, BufferedImage class of Java was used for processing images the applications of BufferedImage class is limited to some operations only, i.e, we can modify the R, G, B values of given input image and produce the modified image. For complex image processing such as face/object detection OpenCV library is used which we will use in this article.

At first we need to setup OpenCV for Java, we recommend to use eclipse for the same since it is easy to use and setup.

Now lets understand some of the methods required for face detection.

1. **CascadeClassifier()** : This class is used to load the trained cascaded set of faces which we will be using to detect face for any input image.
2. **Imcodecs.imread()/Imcodecs.imwrite() :** These methods are used to read and write images as Mat objects which are rendered by OpenCV.
3. **Imgproc.rectangle() :** Used to generate rectangle box outlining faces detected, it takes four arguments – input_image, top_left_point, bottom_right_point, color_of_border.

```
// Java program to demonstrate face detection
package ocv;

import org.opencv.core.Core;
import org.opencv.core.Mat;
import org.opencv.core.MatOfRect;
import org.opencv.core.Point;
import org.opencv.core.Rect;
import org.opencv.core.Scalar;
import org.opencv.imgcodecs.Imgcodecs;
import org.opencv.imgproc.Imgproc;
import org.opencv.objdetect.CascadeClassifier;

public class FaceDetector
{
    public static void main(String[] args)
    {

        // For proper execution of native libraries
        // Core.NATIVE_LIBRARY_NAME must be loaded before
        // calling any of the opencv methods
        System.loadLibrary(Core.NATIVE_LIBRARY_NAME);

        // Face detector creation by loading source cascade xml file
        // using CascadeClassifier.
        // the file can be downloade from
        // https://github.com/opencv/opencv/blob/master/data/haarcascades/
        // haarcascade_frontalface_alt.xml
        // and must be placed in same directory of the source java file
        CascadeClassifier faceDetector = new CascadeClassifier();
        faceDetector.load("haarcascade_frontalface_alt.xml");

        // Input image
        Mat image = Imgcodecs.imread("E:\\input.jpg");

        // Detecting faces
        MatOfRect faceDetections = new MatOfRect();
        faceDetector.detectMultiScale(image, faceDetections);

        // Creating a rectangular box showing faces detected
        for (Rect rect : faceDetections.toArray())
        {
            Imgproc.rectangle(image, new Point(rect.x, rect.y),
             new Point(rect.x + rect.width, rect.y + rect.height),
                                           new Scalar(0, 255, 0));
        }

        // Saving the output image
        String filename = "Ouput.jpg";
        Imgcodecs.imwrite("E:\\"+filename, image);
    }
}
```

Output:

```
    Input.jpg                  Output.jpg
```

input

ouput

**Explanation of Code:**

1. It loads the native OpenCV library to use Java API. An instance of CascadeClassifier is created, passing it the name of the file from which the classifier is loaded.
2. Next, detectMultiScale method is used on the classifier passing it the given image and MatOfRect object.
3. MatOfRect is responsible to do face detections after processing.
4. The process is iterated for doing all the face detections and mark the image with rectangles and at the end image is saved as output.png file.

The output of the program is shown below. This is my pic before and after face detection.
This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

# Image Processing in Java | Set 10 ( Watermarking an image )

We strongly recommend to refer below post as a prerequisite of this.

- Image Processing in Java | Set 1 (Read and Write)
- Image Processing In Java | Set 2 (Get and set Pixels)

In this set we will be generating a watermark and apply it on an input image.
For generating a text and applying it in an image we will use java.awt.Graphics package. Font and color of text is applied by using java.awt.Color and java.awt.Font classes.

Some of the methods used in the code:

- getGraphics() – This method is found in BufferedImage class, and it returns a 2DGraphics object out of image file.
- drawImage(Image img, int x, int y, ImageObserver observer) – The x,y location specifies the position for the top-left of the image. The observer parameter notifies the application of updates to an image that is loaded asynchronously. The observer parameter is not frequently used directly and is not needed for the BufferedImage class, so it usually is null.
- setFont(Font f) – This method is found in Font class of awt package and the constructor takes (FONT_TYPE, FONT_STYLE, FONT_SIZE) as arguments.
- setColor(Color c) – This method is found in Color class of awt package and the constructor takes (R, G, B, A) as arguments.
- drawString(String str, int x, int y) – Fond in Graphics class takes the string text and the location cordinates as x and y as arguments.

```
// Java code for watermarking an image

// For setting color of the watermark text
import java.awt.Color;

// For setting font of the watermark text
import java.awt.Font;

import java.awt.Graphics;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import javax.imageio.ImageIO;

public class WaterMark
{
    public static void main(String[] args)
    {
        BufferedImage img = null;
        File f = null;

        // Read image
        try
        {
           f = new File("input.png");
           img = ImageIO.read(f);
        }
        catch(IOException e)
        {
           System.out.println(e);
        }

        // create BufferedImage object of same width and
        // height as of input image
        BufferedImage temp = new BufferedImage(img.getWidth(),
                    img.getHeight(), BufferedImage.TYPE_INT_RGB);

        // Create graphics object and add original
        // image to it
        Graphics graphics = temp.getGraphics();
        graphics.drawImage(img, 0, 0, null);

        // Set font for the watermark text
        graphics.setFont(new Font("Arial", Font.PLAIN, 80));
        graphics.setColor(new Color(255, 0, 0, 40));

        // Setting watermark text
        String watermark = "WaterMark generated";

        // Add the watermark text at (width/5, height/3)
        // location
        graphics.drawString(watermark, img.getWidth()/5,
                              img.getHeight()/3);

        // releases any system resources that it is using
        graphics.dispose();

        f = new File("output.png");
        try
        {
           ImageIO.write(temp, "png", f);
        }
        catch (IOException e)
        {
           System.out.println(e);
        }
    }
}
```

**NOTE:**Code will not run on online ide since it requires image in drive.

Output:

input.jpg

input

output.jpg

output

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

# Image Processing in Java | Set 11 (Changing orientation of image)

In this article we will use opencv to change orientation of any input image, by using the CORE.flip() method of OpenCV library.

The main idea is that an input buffered image object will be converted to a mat object and then a new mat object will be created in which the original mat object values are put after orientation modification.

For achieving the above result, we will be requiring some of the OpenCV methods:

- **getRaster()** – The method returns a writable raster which in turn is used to get the raw data from input image.
- **put(int row, int column, byte[] data)** / **get(int row, int column, byte[] data)** – used to read/write the raw data into a mat object.
- **flip(mat mat1, mat mat2, int flip_value)** – mat1 and mat2 corresponds to input and output mat objects and the flip_value decides the orientation type.flip_value can be either 0 (flipping along x-axis), 1 (flipping along y-axis), -1 (flipping along both the axis).

```
// Java program to illustrate orientation modification of image
import java.awt.image.BufferedImage;
import java.awt.image.DataBufferByte;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import org.opencv.core.Core;
import org.opencv.core.CvType;
import org.opencv.core.Mat;
```

```
public class OrientingImage
{
  public static void main( String[] args ) throws IOException
  {
    // loads methods of the opencv library
    System.loadLibrary( Core.NATIVE_LIBRARY_NAME );

    // input buffered image object
    File input = new File("E:\\test.jpg");
    BufferedImage image = ImageIO.read(input);

    // converting buffered image object to mat object
    byte[] data = ((DataBufferByte) image.getRaster().getDataBuffer()).getData();
    Mat mat = new Mat(image.getHeight(),image.getWidth(),CvType.CV_8UC3);
    mat.put(0, 0, data);

    // creating a new mat object and putting the modified input mat object by using flip()
    Mat newMat = new Mat(image.getHeight(),image.getWidth(),CvType.CV_8UC3);
    Core.flip(mat, newMat, -1);  //flipping the image about both axis

    // converting the newly created mat object to buffered image object
    byte[] newData = new byte[newMat.rows()*newMat.cols()*(int)(newMat.elemSize())];
    newMat.get(0, 0, newData);
    BufferedImage image1 = new BufferedImage(newMat.cols(), newMat.rows(), 5);
    image1.getRaster().setDataElements(0,0,newMat.cols(),newMat.rows(),newData);

    File ouptut = new File("E:\\result.jpg");
    ImageIO.write(image1, "jpg", ouptut);
  }
}
```

Output:

test.jpg
test

result.jpg
result

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Image-Processing

# Null Pointer Exception In Java

**NullPointerException** is a RuntimeException. In Java, a special null value can be assigned to an object reference. NullPointerException is thrown when program attempts to use an object reference that has the null value.

These can be:

- Invoking a method from a null object.
- Accessing or modifying a null object's field.
- Taking the length of null, as if it were an array.
- Accessing or modifying the slots of null object, as if it were an array.
- Throwing null, as if it were a Throwable value.
- When you try to synchronize over a null object.

**Why do we need the null value?**

Null is a special value used in Java. It is mainly used to indicate that no value is assigned to a reference variable. One application of null is in implementing data structures like linked list and tree. Other applications include Null Object pattern (See this for details) and Singleton pattern. The Singleton pattern ensures that only one instance of a class is created and also, aims for providing a global point of access to the object.

A sample way to create at most one instance of a class is to declare all its constructors as private and then, create a public method that returns the unique instance of the class:

```
// To use randomUUID function.
import java.util.UUID;
import java.io.*;

class Singleton
{
    // Initializing values of single and ID to null.
    private static Singleton single = null;
    private String ID = null;

    private Singleton()
    {
        /* Make it private, in order to prevent the
           creation of new instances of the Singleton
           class. */

        // Create a random ID
        ID = UUID.randomUUID().toString();
    }
    public static Singleton getInstance()
    {
        if (single == null)
            single = new Singleton();
        return single;
    }

    public String getID()
    {
        return this.ID;
    }
}

// Driver Code
public class TestSingleton
{
    public static void main(String[] args)
    {
        Singleton s = Singleton.getInstance();
        System.out.println(s.getID());
    }
}
```

Output:

```
10099197-8c2d-4638-9371-e88c820a9af2
```

In above example, a static instance of the singleton class. That instance is initialized at most once inside the Singleton getInstance method.

**How to avoid the NullPointerException?**

To avoid the NullPointerException, we must ensure that all the objects are initialized properly, before you use them. When we declare a reference variable, we must verify that object is not null, before we request a method or a field from the objects.

Following are the common problems with the solution to overcome that problem.

**Case 1 : String comparison with literals**

A very common case problem involves the comparison between a String variable and a literal. The literal may be a String or an element of an Enum. Instead of invoking the method from the null object, consider invoking it from the literal.

```
// A Java program to demonstrate that invoking a method
// on null causes NullPointerException
import java.io.*;

class GFG
{
    public static void main (String[] args)
    {
        // Initializing String variable with null value
        String ptr = null;

        // Checking if ptr.equals null or works fine.
        try
        {
            // This line of code throws NullPointerException
            // because ptr is null
            if (ptr.equals("gfg"))
                System.out.print("Same");
            else
                System.out.print("Not Same");
        }
        catch(NullPointerException e)
        {
            System.out.print("NullPointerException Caught");
        }
    }
}
```

Output:

```
NullPointerException Caught
```

We can avoid NullPointerException by calling equals on literal rather than object.

```
// A Java program to demonstrate that we can avoid
// NullPointerException
import java.io.*;

class GFG
{
    public static void main (String[] args)
    {
        // Initialing String variable with null value
        String ptr = null;

        // Checking if ptr is null using try catch.
        try
        {
            if ("gfg".equals(ptr))
                System.out.print("Same");
            else
                System.out.print("Not Same");
        }
        catch(NullPointerException e)
        {
            System.out.print("Caught NullPointerException");
        }
    }
}
```

Output:

```
Not Same
```

**Case 2 : Keeping a Check on the arguments of a method**

Before executing the body of your new method, we should first check its arguments for null values and continue with execution of the method, only when the arguments are properly checked. Otherwise, it will throw an **IllegalArgumentException** and notify the calling method that something is wrong with the passed arguments.

```
// A Java program to demonstrate that we should
// check if parameters are null or not before
// using them.
import java.io.*;

class GFG
{
    public static void main (String[] args)
    {
        // String s set an empty string  and calling getLength()
        String s = "";
        try
        {
            System.out.println(getLength(s));
        }
        catch(IllegalArgumentException e)
        {
            System.out.println("IllegalArgumentException caught");
        }

        // String s set to a value and calling getLength()
        s = "GeeksforGeeks";
        try
        {
            System.out.println(getLength(s));
        }
        catch(IllegalArgumentException e)
        {
            System.out.println("IllegalArgumentException caught");
        }

        // Setting s as null and calling getLength()
        s = null;
        try
        {
            System.out.println(getLength(s));
        }
        catch(IllegalArgumentException e)
        {
            System.out.println("IllegalArgumentException caught");
        }
    }

    // Function to return length of string s. It throws
    // IllegalArgumentException if s is null.
    public static int getLength(String s)
    {
        if (s == null)
            throw new IllegalArgumentException("The argument cannot be null");
        return s.length();
    }
}
```

Output:

```
0
13
IllegalArgumentException caught
```

**Case 3 : Use of Ternary Operator**

The ternary operator can be used to avoid NullPointerException. First, the Boolean expression is evaluated. If the expression is **true** then, the value1 is returned, otherwise, the value2 is returned. We can use the ternary operator for handling null pointers:

```
// A Java program to demonstrate that we can use
// ternary operator to avoid NullPointerException.
import java.io.*;

class GFG
{
    public static void main (String[] args)
    {
        // Initializing String variable with null value
        String str = null;
        String message = (str == null) ? "" :
                      str.substring(0,5);
        System.out.println(message);

        // Initializing String variable with null value
        str = "Geeksforgeeks";
        message = (str == null) ? "" : str.substring(0,5);
        System.out.println(message);
    }
}
```

Output:

```
Geeks
```

The message variable will be empty if str's reference is **null** as in case 1. Otherwise, if str point to **actual data**, the message will retrieve the first 6 characters of it as in case 2.

Related Article – Interesting facts about Null in Java

This article is contributed by **Nikhil Meherwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Technical Scripter
Exception Handling

# enum in Java

Enumerations serve the purpose of representing a group of named constants in a programming language. For example the 4 suits in a deck of playing cards may be 4 enumerators named Club, Diamond, Heart, and

Spade, belonging to an enumerated type named Suit. Other examples include natural enumerated types (like the planets, days of the week, colors, directions, etc.).
Enums are used when we know all possible values at **compile time**, such as choices on a menu, rounding modes, command line flags, etc. It is not necessary that the set of constants in an enum type stay **fixed** for all time.

In Java (from 1.5), enums are represented using **enum** data type. Java enums are more powerful than C/C++ enums . In Java, we can also add variables, methods and constructors to it. The main objective of enum is to define our own data types(Enumerated Data Types).

**Declaration of enum in java :**

- Enum declaration can be done outside a Class or inside a Class but not inside a Method.

```
// A simple enum example where enum is declared
// outside any class (Note enum keyword instead of
// class keyword)
enum Color
{
    RED, GREEN, BLUE;
}

public class Test
{
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output :

```
RED
```

```
// enum declaration inside a class.
public class Test
{
    enum Color
    {
        RED, GREEN, BLUE;
    }

    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output :

```
RED
```

- First line inside enum should be list of constants and then other things like methods, variables and constructor.
- According to Java naming conventions, it is recommended that we name constant with all capital letters

**Important points of enum :**

- Every enum internally implemented by using Class.

```
/* internally above enum Color is converted to
class Color
{
     public static final Color RED = new Color();
     public static final Color BLUE = new Color();
     public static final Color GREEN = new Color();
}*/
```

- Every enum constant represents an **object** of type enum.
- enum type can be passed as an argument to **switch** statement.

```
// A Java program to demonstrate working on enum
// in switch case (Filename Test. Java)
import java.util.Scanner;

// An Enum class
enum Day
{
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY;
}

// Driver class that contains an object of "day" and
// main().
public class Test
{
    Day day;

    // Constructor
    public Test(Day day)
    {
        this.day = day;
    }

    // Prints a line about Day using switch
    public void dayIsLike()
    {
        switch (day)
        {
        case MONDAY:
            System.out.println("Mondays are bad.");
            break;
        case FRIDAY:
            System.out.println("Fridays are better.");
            break;
        case SATURDAY:
        case SUNDAY:
            System.out.println("Weekends are best.");
            break;
        default:
            System.out.println("Midweek days are so-so.");
            break;
        }
    }

    // Driver method
    public static void main(String[] args)
    {
        String str = "MONDAY";
        Test t1 = new Test(Day.valueOf(str));
        t1.dayIsLike();
    }
}
```

Output:

```
Mondays are bad.
```

- Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using enum Name. Since it is **final**, we can't create child enums.
- We can declare **main() method** inside enum. Hence we can invoke enum directly from the Command Prompt.

```
// A Java program to demonstrate that we can have
// main() inside enum class.
enum Color
{
    RED, GREEN, BLUE;

    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
    }
}
```

Output :

```
RED
```

**Enum and Inheritance :**

- All enums implicitly extend **java.lang.Enum class**. As a class can only extend **one** parent in Java, so an enum cannot extend anything else.
- **toString() method** is overridden in **java.lang.Enum class**,which returns enum constant name.
- enum can implement many interfaces.

**values(), ordinal() and valueOf() methods :**

- These methods are present inside **java.lang.Enum**.
- **values() method** can be used to return all values present inside enum.
- Order is important in enums.By using **ordinal() method**, each enum constant index can be found, just like array index.
- **valueOf() method** returns the enum constant of the specified string value, if exists.

```
// Java program to demonstrate working of values(),
// ordinal() and valueOf()
enum Color
{
    RED, GREEN, BLUE;
}

public class Test
{
    public static void main(String[] args)
    {
        // Calling values()
        Color arr[] = Color.values();

        // enum with loop
        for (Color col : arr)
        {
            // Calling ordinal() to find index
            // of color.
            System.out.println(col + " at index "
                             + col.ordinal());
        }

        // Using valueOf(). Returns an object of
        // Color with given constant.
        // Uncommenting second line causes exception
        // IllegalArgumentException
        System.out.println(Color.valueOf("RED"));
        // System.out.println(Color.valueOf("WHITE"));
    }
}
```

Output :

```
RED at index 0
GREEN at index 1
BLUE at index 2
RED
```

**enum and constructor :**

- enum can contain constructor and it is executed separately for each enum constant at the time of enum class loading.
- We can't create enum objects explicitly and hence we can't invoke enum constructor directly.

**enum and methods :**

- enum can contain **concrete** methods only i.e. no any **abstract** method.

```
// Java program to demonstrate that enums can have constructor
// and concrete methods.

// An enum (Note enum keyword inplace of class keyword)
enum Color
{
    RED, GREEN, BLUE;
    // enum constructor called separately for each
    // constant
    private Color()
    {
        System.out.println("Constructor called for : " +
        this.toString());
    }

    // Only concrete (not abstract) methods allowed
    public void colorInfo()
    {
        System.out.println("Universal Color");
    }
}

public class Test
{
    // Driver method
    public static void main(String[] args)
    {
        Color c1 = Color.RED;
        System.out.println(c1);
        c1.colorInfo();
    }
}
```

Output:

```
Constructor called for : RED
Constructor called for : GREEN
Constructor called for : BLUE
RED
Universal Color
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# transient keyword in Java

**transient** is a variables modifier used in serialization. At the time of serialization, if we don't want to save value of a particular variable in a file, then we use **transient** keyword. When JVM comes across **transient** keyword, it ignores original value of the variable and save default value of that variable data type.

**transient** keyword plays an important role to meet security constraints. There are various real-life examples where we don't want to save private data in file. Another use of **transient** keyword is not to serialize the variable whose value can be calculated/derived using other serialized objects or system such as age of a person, current date, etc.
Practically we serialized only those fields which represent a state of instance, after all serialization is all about to save state of an object to a file. It is good habit to use **transient** keyword with private confidential fields of a class during serialization.

```
// A sample class that uses transient keyword to
// skip their serialization.
class Test implements Serializable
{
    // Making password transient for security
    private transient String password;

    // Making age transient as age is auto-
    // computable from DOB and current date.
    transient int age;

    // serialize other fields
    private String username, email;
    Date dob;

    // other code
}
```

**transient and static :** Since **static** fields are not part of state of the object, there is no use/impact of using **transient** keyword with static variables. However there is no compilation error.

**transient and final :** final variables are directly serialized by their values, so there is no use/impact of declaring final variable as **transient**. There is no compile-time error though.

```
// Java program to demonstrate transient keyword
// Filename Test.java
import java.io.*;
class Test implements Serializable
{
    // Normal variables
    int i = 10, j = 20;

    // Transient variables
    transient int k = 30;

    // Use of transient has no impact here
    transient static int l = 40;
    transient final int m = 50;

    public static void main(String[] args) throws Exception
    {
        Test input = new Test();

        // serialization
        FileOutputStream fos = new FileOutputStream("abc.txt");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(input);

        // de-serialization
        FileInputStream fis = new FileInputStream("abc.txt");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Test output = (Test)ois.readObject();
        System.out.println("i = " + output.i);
        System.out.println("j = " + output.j);
        System.out.println("k = " + output.k);
        System.out.println("l = " + output.l);
        System.out.println("m = " + output.m);
    }
}
```

Output :

```
i = 10
j = 20
k = 0
l = 40
m = 50
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Interesting facts about null in Java

Almost all the programming languages are bonded with null. There is hardly a programmer, who is not troubled by null.
In Java, null is associated java.lang.NullPointerException. As it is a class in java.lang package, it is called when we try to perform some operations with or without null and sometimes we don't even know where it has happened.

Below are some important points about null in java which every Java programmer should know:

**1. null is Case sensitive:** null is keyword in java and because keywords are **case-sensitive** in java, we can't write NULL or 0 as in C language.

```
public class Test
{
 public static void main (String[] args) throws java.lang.Exception
 {
     // compile-time error : can't find symbol 'NULL'
     Object obj = NULL;

  //runs successfully
  Object obj1 = null;
    }
}
```

Output:

```
5: error: cannot find symbol
 can't find symbol 'NULL'
              ^
   variable NULL
 class Test
1 error
```

**2. Reference Variable value:** Any reference variable in Java has default value null.

```
public class Test
{
    private static Object obj;
    public static void main(String args[])
    {
        // it will print null;
        System.out.println("Value of object obj is : " + obj);
    }
}
```

Output:

```
Value of object obj is : null
```

**3. Type of null:** Unlike common misconception, null is not Object or neither a type. It's just a special value, which can be assigned to any reference type and you can type cast null to any type
Examples:

```
    // null can be assigned to String
    String str = null;

    // you can assign null to Integer also
    Integer itr = null;

    // null can also be assigned to Double
    Double dbl = null;

    // null can be type cast to String
    String myStr = (String) null;

    // it can also be type casted to Integer
    Integer myItr = (Integer) null;

    // yes it's possible, no error
    Double myDbl = (Double) null;
```

**4. Autoboxing and unboxing :** During auto-boxing and unboxing operations, compiler simply throws Nullpointer exception error if a null value is assigned to primitive boxed data type.

```
public class Test
{
 public static void main (String[] args) throws java.lang.Exception
 {
   //An integer can be null, so this is fine
   Integer i = null;

   //Unboxing null to integer throws NullpointerException
   int a = i;
 }
}
```

Output:

```
 Exception in thread "main" java.lang.NullPointerException
 at Test.main(Test.java:6)
```

**5. instanceof operator:** The java instanceof operator is used to test whether the object is an instance of the specified type (class or subclass or interface). At run time, the result of the instanceof operator is true if the value of the Expression is not null.
This is an important property of instanceof operation which makes it useful for type casting checks.

```
public class Test
{
 public static void main (String[] args) throws java.lang.Exception
 {
  Integer i = null;
  Integer j = 10;

  //prints false
  System.out.println(i instanceof Integer);

  //Compiles successfully
  System.out.println(j instanceof Integer);
 }
}
```

Output:

```
false
true
```

**6. Static vs Non static Methods:** We cannot call a non-static method on a reference variable with null value, it will throw NullPointerException, but we can call static method with reference variables with null values. Since static methods are bonded using static binding, they won't throw Null pointer Exception.

```
public class Test
{
    public static void main(String args[])
    {
        Test obj= null;
        obj.staticMethod();
        obj.nonStaticMethod();
    }

 private static void staticMethod()
 {
  //Can be called by null reference
  System.out.println("static method, can be called by null reference");

 }

 private void nonStaticMethod()
 {
     //Can not be called by null reference
     System.out.print(" Non-static method- ");
     System.out.println("cannot be called by null reference");

 }

}
```

Output:

```
static method, can be called by null referenceException in thread "main"
java.lang.NullPointerException
 at Test.main(Test.java:5)
```

**7. == and !=** The comparision and not equal to operators are allowed with null in Java. This can made useful in checking of null with objects in java.

```
public class Test
{
    public static void main(String args[])
    {

    //return true;
    System.out.println(null==null);

    //return false;
    System.out.println(null!=null);

 }
}
```

Output:

```
true
false
```

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Java

# Establishing JDBC Connection in Java

Before establishing a connection between front end i.e your Java Program and back end i.e the database we should learn what precisely a JDBC is and why it came to existence.

**What is JDBC ?**

JDBC is an acronym for Java Database Connectivity. It's an advancement for ODBC ( Open Database Connectivity ). JDBC is an standard API specification developed in order to move data from frontend to backend. This API consists of classes and interfaces written in Java. It basically acts as an interface (not the one we use in Java) or channel between your Java program and databases i.e it establishes a link between the two so that a programmer could send data from Java code and store it in the database for future use.

**Why JDBC came into existence ?**

As previously told JDBC is an advancement for ODBC, ODBC being platform dependent had a lot of drawbacks. ODBC API was written in C,C++, Python, Core Java and as we know above languages (except Java and some part of Python )are platform dependent . Therefore to remove dependence, JDBC was developed by database vendor which consisted of classes and interfaces written in Java.

**Steps for connectivity between Java program and database**

**1. Loading the Driver**
To begin with, you first need load the driver or register it before using it in the program . Registration is to be done once in your program. You can register a driver in one of two ways mentioned below :

- **Class.forName() :** Here we load the driver's class file into memory at the runtime. No need of using new or creation of object .The following example uses Class.forName() to load the Oracle driver –

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

- **DriverManager.registerDriver():** DriverManager is a Java inbuilt class with a static member register. Here we call the constructor of the driver class at compile time . The following example uses DriverManager.registerDriver()to register the Oracle driver –

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver())
```

2. **Create the connections**
After loading the driver, establish connections using :

```
Connection con = DriverManager.getConnection(url,user,password)
```

**user** – username from which your sql command prompt can be accessed.
**password** – password from which your sql command prompt can be accessed.

**con:** is a reference to Connection interface.
**url** : Uniform Resource Locator. It can be created as follows:

```
String url = " jdbc:oracle:thin:@localhost:1521:xe"
```

Where oracle is the database used, thin is the driver used , @localhost is the IP Address where database is stored, 1521 is the port number and xe is the service provider. All 3 parameters above are of String type and are to be declared by programmer before calling the function. Use of this can be referred from final code.

**3. Create a statement**
Once a connection is established you can interact with the database. The JDBCStatement, CallableStatement, and PreparedStatement interfaces define the methods that enable you to send SQL commands and receive data from your database.
Use of JDBC Statement is as follows:

```
Statement st = con.createStatement();
```

Here, con is a reference to Connection interface used in previous step .

**4. Execute the query**
Now comes the most important part i.e executing the query. Query here is an SQL Query . Now we know we can have multiple types of queries. Some of them are as follows:

- Query for updating / inserting table in a database.
- Query for retrieving data .

The executeQuery() method of Statement interface is used to execute queries of retrieving values from the database. This method returns the object of ResultSet that can be used to get all the records of a table.
The executeUpdate(sql query) method ofStatement interface is used to execute queries of updating/inserting .

Example:

```
int m = st.executeUpdate(sql);
if (m==1)
   System.out.println("inserted successfully : "+sql);
else
   System.out.println("insertion failed");
```

Here sql is sql query of the type String

**5.Close the connections**
So finally we have sent the data to the specified location and now we are at the verge of completion of our task .
By closing connection, objects of Statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.
Example :

```
con.close();
```

**Implementation**

```
importjava.sql.*;
importjava.util.*;
class Main
{
   public static void main(String a[])
   {
      //Creating the connection
      String url = "jdbc:oracle:thin:@localhost:1521:xe";
      String user = "system";
      String pass = "12345";

      //Entering the data
      Scanner k = new Scanner(System.in);
      System.out.println("enter name");
      String name = k.next();
      System.out.println("enter roll no");
      int roll = k.nextInt();
      System.out.println("enter class");
      String cls =  k.next();

      //Inserting data using SQL query
      String sql = "insert into student1 values('"+name+"',"+roll+",'"+cls+"')";
      Connection con=null;
      try
      {
         DriverManager.registerDriver(new oracle.jdbc.OracleDriver());

         //Reference to connection interface
         con = DriverManager.getConnection(url,user,pass);

         Statement st = con.createStatement();
         int m = st.executeUpdate(sql);
         if (m == 1)
            System.out.println("inserted successfully : "+sql);
         else
            System.out.println("insertion failed");
         con.close();
      }
      catch(Exception ex)
```

```
        {
            System.err.println(ex);
        }
    }
}
```

Output –

You may also see – What is JDBC Connection? Explain steps to get Database connection in a simple java program.

This article is contributed by **Shreya Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
JDBC

## Does Java support goto?

Unlike C/C++, Java does not have *goto* statement, but java supports label. The only place where a label is useful in Java is right before nested loop statements. We can specify label name with break to break out a specific outer loop. Similarly, label name can be specified with continue.

See following program for example.

```
// file name: Main.java
public class Main {
  public static void main(String[] args) {
    outer: //label for outer loop
    for (int i = 0; i < 10; i++) {
      for (int j = 0; j < 10; j++) {
        if (j == 1)
          break outer;
        System.out.println(" value of j = " + j);
      }
    } //end of outer loop
  } // end of main()
} //end of class Main
```

Output:
*value of j = 0*

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

## Inner class in java

Inner class means one class which is a member of another class. There are basically four types of inner classes in java.

1) Nested Inner class
2) Method Local inner classes
3) Anonymous inner classes
4) Static nested classes

**Nested Inner class** can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier.
Like class, interface can also be nested and can have access specifiers.

Following example demonstrates a nested class.

```
class Outer {
   // Simple nested inner class
   class Inner {
      public void show() {
           System.out.println("In a nested class method");
      }
   }
}
class Main {
   public static void main(String[] args) {
       Outer.Inner in = new Outer().new Inner();
       in.show();
   }
}
```

Output:

```
In a nested class method
```

As a side note, we can't have static method in a nested inner class because an inner class is implicitly associated with an object of its outer class so it cannot define any static method for itself. For example the following program doesn't compile.

```
class Outer {
   void outerMethod() {
      System.out.println("inside outerMethod");
   }
   class Inner {
    public static void main(String[] args){
      System.out.println("inside inner class Method");
    }
  }
}
```

Output:

```
Error illegal static declaration in inner class
Outer.Inner public static void main(String[] args)
modifier ‘static’ is only allowed in constant
variable declaration
```

An interface can also be nested and nested interfaces have some interesting properties. We will be covering nested interfaces in the next post.

**Method Local inner classes**

Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

```
class Outer {
    void outerMethod() {
        System.out.println("inside outerMethod");
        // Inner class is local to outerMethod()
        class Inner {
            void innerMethod() {
                System.out.println("inside innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
class MethodDemo {
    public static void main(String[] args) {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Output

```
Inside outerMethod
Inside innerMethod
```

Method Local inner classes can’t use local variable of outer method until that local variable is not declared as final. For example, the following code generates compiler error (Note that x is not final in outerMethod() and innerMethod() tries to access it)

```
class Outer {
   void outerMethod() {
      int x = 98;
      System.out.println("inside outerMethod");
      class Inner {
         void innerMethod() {
            System.out.println("x= "+x);
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodLocalVariableDemo {
   public static void main(String[] args) {
      Outer x=new Outer();
      x.outerMethod();
   }
}
```

Output:

```
local variable x is accessed from within inner class;
needs to be declared final
```

But the following code compiles and runs fine (Note that x is final this time)

```
class Outer {
   void outerMethod() {
      final int x=98;
      System.out.println("inside outerMethod");
      class Inner {
         void innerMethod() {
            System.out.println("x = "+x);
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodLocalVariableDemo {
    public static void main(String[] args){
      Outer x = new Outer();
      x.outerMethod();
    }
}
```

Output-:

```
Inside outerMethod
X = 98
```

The main reason we need to declare a local variable as a final is that local variable lives on stack till method is on the stack but there might be a case the object of inner class still lives on the heap.
Method local inner class can’t be marked as private, protected, static and transient but can be marked as abstract and final, but not both at the same time.

**Static nested classes**

Static nested classes are not technically an inner class. They are like a static member of outer class.

```
class Outer {
   private static void outerMethod() {
       System.out.println("inside outerMethod");
   }

   // A static inner class
   static class Inner {
       public static void main(String[] args) {
           System.out.println("inside inner class Method");
           outerMethod();
       }
   }

}
```

Output

```
inside inner class Method
inside outerMethod
```

**Anonymous inner classes**

Anonymous inner classes are declared without any name at all. They are created in two ways.

***a)** As subclass of specified type*

```
class Demo {
    void show() {
        System.out.println("i am in show method of super class");
    }
}
class Flavor1Demo {

    //  An anonymous class with Demo as base class
    static Demo d = new Demo() {
        void show() {
            super.show();
            System.out.println("i am in Flavor1Demo class");
        }
    };
    public static void main(String[] args){
        d.show();
    }
}
```

Output

```
i am in show method of super class
i am in Flavor1Demo class
```

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.

***a)** As implementer of the specified interface*

```
class Flavor2Demo {

    // An anonymous class that implements Hello interface
    static Hello h = new Hello() {
        public void show() {
            System.out.println("i am in anonymous class");
        }
    };

    public static void main(String[] args) {
        h.show();
    }
}

interface Hello {
    void show();
}
```

Output:

```
i am in anonymous class
```

In above code we create an object of anonymous inner class but this anonymous inner class is an implementer of the interface Hello. Any anonymous inner class can implement only one interface at one time. It can either extend a class or implement interface at a time.

This article is contributed by **Pawan Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java
Java

# The Initializer Block in Java

Initializer block contains the code that is always executed whenever an instance is created. It is used to declare/initialize the common part of various constructors of a class. For example,

```
import java.io.*;
public class GFG
{
    // Initializer block starts..
    {
        // This code is executed before every constructor.
        System.out.println("Common part of constructors invoked !!");
    }
    // Initializer block ends

    public GFG()
    {
        System.out.println("Default Constructor invoked");
    }
    public GFG(int x)
    {
        System.out.println("Parametrized constructor invoked");
    }
    public static void main(String arr[])
    {
        GFG obj1, obj2;
        obj1 = new GFG();
        obj2 = new GFG(0);
    }
}
```

**Output:**

```
Common part of constructors invoked!!
Default Constructor invoked
Common part of constructors invoked!!
Parametrized constructor invoked
```

We can note that the contents of initializer block are executed whenever any constructor is invoked (before the constructor's contents)

The order of initialization constructors and initializer block doesn't matter, initializer block is always executed before constructor. See this for example.

**What if we want to execute some code once for all objects of a class?**
We use Static Block in Java

This article is contributed by **Ashutosh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Few Tricky Programs in Java

1. **Comments that execute :**

   Till now, we were always taught "Comments do not Execute". Let us see today "The comments that execute"

   Following is the code snippet:

```
public class Testing {
    public static void main(String[] args)
     {
         // the line below this gives an output
         // \u000d System.out.println("comment executed");
     }
}
```

   Output:

```
comment executed
```

   The reason for this is that the Java compiler parses the unicode character \u000d as a new line and gets transformed into:

```
public class Testing {
    public static void main(String[] args)
    {
        // the line below this gives an output
        // \u000d
        System.out.println("comment executed");
    }
}
```

2. **Named loops :**

```
// A Java program to demonstrate working of named loops.
public class Testing
{
    public static void main(String[] args)
    {
    loop1:
    for (int i = 0; i < 5; i++)
     {
        for (int j = 0; j < 5; j++)
        {
            if (i == 3)
                break loop1;
            System.out.println("i = " + i + " j = " + j);
        }
    }
   }
}
```

   Output:

```
i = 0 j = 0
i = 0 j = 1
i = 0 j = 2
i = 0 j = 3
i = 0 j = 4
i = 1 j = 0
i = 1 j = 1
i = 1 j = 2
i = 1 j = 3
i = 1 j = 4
i = 2 j = 0
i = 2 j = 1
i = 2 j = 2
i = 2 j = 3
i = 2 j = 4
```

   You can also use continue to jump to start of the named loop.

   We can also use break (or continue) in a nested if-else with for loops in order to break several loops with if-else, so one can avoid setting lot of flags and testing them in the if-else in order to continue or not in this nested level.

This article is contributed by **Abhineet Nigam**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Foreach in C++ and Java

Foreach loop is used to access elements of an array quickly without performing initialization, testing and increment/decrement. The working of foreach loops is to do something for every element rather than doing something n times.

There is no foreach loop in C, but both C++ and Jave support foreach type of loop. In C++, it was introduced in C++ 11 and Java in JDK 1.5.0

The keyword used for foreach loop is “**for**” in both C++ and Java.

**C++ Program:**

```
// C++ program to demonstrate use of foreach
#include <iostream>
using namespace std;

int main()
{
  int arr[] = {10, 20, 30, 40};

  // Printing elements of an array using
  // foreach loop
  for (int x : arr)
     cout << x << endl;
}
```

Output:

```
10
20
30
40
```

**Java program**

```
// Java program to demonstrate use of foreach
public class Main
{
    public static void main(String[] args)
    {
       // Declaring 1-D array with size 4
       int arr[] = {10, 20, 30, 40};

       // Printing elements of an array using
       // foreach loop
       for (int x : arr)
          System.out.println(x);
    }
}
```

Output:

```
10
20
30
40
```

Advantages of Foreach loop :-
1) Makes code more readable.
2) Eliminates the possibility of programming errors.

This article is contributed by **Rahul Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# Flexible nature of java.lang.Object

We all love the mechanism of python, where we don’t have to bother about data types of the variables (don’t we!)

Interestingly we have one class in Java too, which is pretty similar !

Yes, you guessed it right! It’s java.lang.Object

For example,

```
// A Java program to demonstrate flexible nature of
// java.lang.Object
public class GFG
{
    public static void main(String arr[])
    {
        Object y;

        y = 'A';
        System.out.println(y.getClass().getName());

        y = 1;
        System.out.println(y.getClass().getName());

        y = "Hi";
        System.out.println(y.getClass().getName());

        y = 1.222;
        System.out.println(y.getClass().getName());

        y = false;
        System.out.println(y.getClass().getName());
    }
}
```

**Output:**

```
java.lang.Character
java.lang.Integer
java.lang.String
java.lang.Double
java.lang.Boolean
```

Such a behaviour can be attributed to the fact that java.lang.Object is super class to all other classes. Hence, a reference variable of type Object can be practically used to refer objects of any class. So, we could also assign y = new InputStreamReader(System.in) in the above code!

This article is contributed by **Ashutosh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# How to swap or exchange objects in Java?

**How to swap objects in Java?**

Let's say we have a class called "Car" with some attributes. And we create two objects of Car, say car1 and car2, how to exchange the data of car1 and car2?

**A Simple Solution is to swap members.** For example, if the class Car has only one integer attribute say "no" (car number), we can swap cars by simply swapping the members of two cars.

```
// A Java program to demonstrate that we can swap two
// objects be swapping members.

// A car with number class Car
class Car
{
    int no;
    Car(int no) { this.no = no; }
}

// A class that uses Car objects
class Main
{
    // To swap c1 and c2
    public static void swap(Car c1, Car c2)
    {
        int temp = c1.no;
        c1.no = c2.no;
        c2.no = temp;
    }

    // Driver method
    public static void main(String[] args)
    {
        Car c1 = new Car(1);
        Car c2 = new Car(2);
        swap(c1, c2);
        System.out.println("c1.no = " + c1.no);
        System.out.println("c2.no = " + c2.no);
    }
}
```

Output:

```
c1.no = 2
c2.no = 1
```

**What if we don't know members of Car?**

The above solution worked as we knew that there is one member "no" in Car. What if we don't know members of Car or the member list is too big. T*his is a very common situation as a class that uses some other class may not access members of other class.* Does below solution work?

```
// A Java program to demonstrate that simply swapping
// object references doesn't work

// A car with number and name
class Car
{
    int model, no;

    // Constructor
    Car(int model, int no)
    {
        this.model = model;
        this.no = no;
    }

    // Utility method to print Car
    void print()
    {
        System.out.println("no = " + no +
                           ", model = " + model);
    }
}

// A class that uses Car
class Main
{
    // swap() doesn't swap c1 and c2
    public static void swap(Car c1, Car c2)
    {
        Car temp = c1;
        c1 = c2;
        c2 = temp;
    }

    // Driver method
    public static void main(String[] args)
    {
        Car c1 = new Car(101, 1);
        Car c2 = new Car(202, 2);
        swap(c1, c2);
        c1.print();
        c2.print();
    }
}
```

Output:

```
no = 1, model = 101
no = 2, model = 202
```

As we can see from above output, the objects are not swapped. We have discussed in a previous post that parameters are passed by value in Java. So when we pass c1 and c2 to swap(), the function swap() creates a copy of these references.

**Solution is to use Wrapper Class** If we create a wrapper class that contains references of Car, we can swap cars by swapping references of wrapper class.

```
// A Java program to demonstrate that we can use wrapper
// classes to swap to objects

// A car with model and no.
class Car
{
    int model, no;

    // Constructor
    Car(int model, int no)
    {
        this.model = model;
        this.no = no;
    }

    // Utility method to print object details
    void print()
    {
        System.out.println("no = " + no +
                           ", model = " + model);
    }
}

// A Wrapper over class that is used for swapping
class CarWrapper
{
   Car c;

   // Constructor
   CarWrapper(Car c)   {this.c = c;}
}

// A Class that use Car and swaps objects of Car
// using CarWrapper
class Main
{
    // This method swaps car objects in wrappers
    // cw1 and cw2
    public static void swap(CarWrapper cw1,
                            CarWrapper cw2)
    {
        Car temp = cw1.c;
        cw1.c = cw2.c;
        cw2.c = temp;
    }

    // Driver method
    public static void main(String[] args)
    {
        Car c1 = new Car(101, 1);
        Car c2 = new Car(202, 2);
        CarWrapper cw1 = new CarWrapper(c1);
        CarWrapper cw2 = new CarWrapper(c2);
        swap(cw1, cw2);
        cw1.c.print();
        cw2.c.print();
    }
}
```

Output:

```
no = 2, model = 202
no = 1, model = 101
```

So a wrapper class solution works even if the user class doesn't have access to members of the class whose objects are to be swapped.

This article is contributed by **Anurag Rai**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java

## How to swap two variables in one line in C/C++, Python and Java?

We have discussed different approaches to swap two integers without the temporary variable. How to swap in a single line without using library function?

**Python:** In Python, there is a simple and syntactically neat construct to swap variables, we just need to write "x, y = y, x".

**C/C++:** Below is one generally provided classical solution

```
// Swap using bitwise XOR (Wrong Solution in C/C++)
x ^= y ^= x ^= y;
```

The above solution is wrong in C/C++ as it causes undefined behaviour (compiler is free to behave in any way). The reason is, modifying a variable more than once in an expression causes undefined behaviour if there is no sequence point between the modifications.

However, we can use comma to introduce sequence points. So the modified solution is

```
// Swap using bitwise XOR (Correct Solution in C/C++)
// sequence point introduced using comma.
(x ^= y), (y ^= x), (x ^= y);
```

**Java:** In Java, rules for subexpression evaluations are clearly defined. The left hand operand is always evaluated before right hand operand (See this for more details). In Java, the expression "x ^= y ^= x ^= y;" doesn't produce the correct result according to Java rules. It makes x = 0. However, we can use "x = x ^ y ^ (y = x);" Note the expressions are evaluated from left to right. If x = 5 and y = 10 initially, the expression is equivalent to "x = 5 ^ 10 ^ (y = 5);". Note that we can't use this in C/C++ as in C/C++, it is not defined whether left operand or right operand is executed for any operator (See this for more details)

### C/C++

```
// C/C++ program to swap two variables in single line
#include <stdio.h>
int main()
{
    int x = 5, y = 10;
    (x ^= y), (y ^= x), (x ^= y);
    printf("After Swapping values of x and y are %d %d",
            x, y);
    return 0;
}
```

### Java

```
// Java program to swap two variables in single line
class GFG
{
   public static void main (String[] args)
   {
      int x = 5, y = 10;
      x = x ^ y ^ (y = x);
      System.out.println("After Swapping values of x and y are "
                         + x + " " + y);
   }
}
```

### Python

```
# Python program to swap two variables in single line
x = 5
y = 10
x, y = y, x
print "After Swapping values of x and y are", x, y
```

Output:

```
After Swapping values of x and y are 10 5
```

This article is contributed by **Harshit Gupta.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

C/C++ Puzzles
Java
Python
XOR

# Private Constructors and Singleton Classes in Java

Let's first analyze the following question:

***Can we have private constructors ?***

As you can easily guess, like any method we can provide access specifier to the constructor. If it's made private, then it can only be accessed inside the class.

***Do we need such 'private constructors ' ?***

There are various scenarios where we can use private constructors. The major ones are

1. Internal Constructor chaining
2. Singleton class design pattern

***What is a Singleton class?***

As the name implies, a class is said to be singleton if it limits the number of objects of that class to one.

We can't have more than a single object for such classes.

Singleton classes are employed extensively in concepts like Networking and Database Connectivity.

**Design Pattern of Singleton classes:**

The constructor of singleton class would be private so there must be another way to get the instance of that class. This problem is resolved using a class member instance and a factory method to return the class member.

Below is an example in java illustrating the same:

```
// Java program to demonstrate implementation of Singleton
// pattern using private constructors.
import java.io.*;

class MySingleton
{
    static MySingleton instance = null;
    public int x = 10;

    // private constructor can't be accessed outside the class
    private MySingleton() {  }

    // Factory method to provide the users with instances
    static public MySingleton getInstance()
    {
        if (instance == null)
            instance = new MySingleton();

        return instance;
    }
}

// Driver Class
class Main
{
   public static void main(String args[])
   {
       MySingleton a = MySingleton.getInstance();
       MySingleton b = MySingleton.getInstance();
       a.x = a.x + 10;
       System.out.println("Value of a.x = " + a.x);
       System.out.println("Value of b.x = " + b.x);
   }
}
```

Output:

```
Value of a.x = 20
Value of b.x = 20
```

We changed value of a.x, value of b.x also got updated because both 'a' and 'b' refer to same object, i.e., they are objects of a singleton class.

Refer this for implementation of same in C++.

This article is contributed by **Ashutosh Kumar Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Java
Java

# Generics in Java

Generics in Java is similar to templates in C++. The idea is to allow type (Integer, String, … etc and user defined types) to be a parameter to methods, classes and interfaces. For example, classes like HashSet, ArrayList, HashMap, etc use generics very well. We can use them for any type.

**Generic Class**

Like C++, we use <> to specify parameter types in generic class creation. To create objects of generic class, we use following syntax.

```
// To create an instance of generic class
BaseType <Type> obj = new BaseType <Type>()

Note: In Parameter type we can not use primitives like
      'int','char' or 'double'.
```

```
// A Simple Java program to show working of user defined
// Generic classes

// We use < > to specify Parameter type
class Test<T>
{
    // An object of type T is declared
    T obj;
    Test(T obj) {  this.obj = obj;  }  // constructor
    public T getObject()  { return this.obj; }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        // instance of Integer type
        Test <Integer> iObj = new Test<Integer>(15);
        System.out.println(iObj.getObject());

        // instance of String type
        Test <String> sObj =
                          new Test<String>("GeeksForGeeks");
        System.out.println(sObj.getObject());
    }
}
```

Output:

```
15
GeeksForGeeks
```

We can also pass multiple Type parameters in Generic classes.

```
// A Simple Java program to show multiple
// type parameters in Java Generics

// We use < > to specify Parameter type
class Test<T, U>
{
    T obj1;  // An object of type T
    U obj2;  // An object of type U

    // constructor
    Test(T obj1, U obj2)
    {
        this.obj1 = obj1;
        this.obj2 = obj2;
    }
    // To print objects of T and U
    public void print()
    {
        System.out.println(obj1);
        System.out.println(obj2);
    }
}

// Driver class to test above
class Main
{
    public static void main (String[] args)
    {
        Test <String, Integer> obj =
            new Test<String, Integer>("GfG", 15);

        obj.print();
    }
}
```

Output:

```
GfG
15
```

**Generic Functions:**

We can also write generic functions that can be called with different types of arguments based on the type of arguments passed to generic method, the compiler handles each method.

```
// A Simple Java program to show working of user defined
// Generic functions

class Test
{
    // A Generic method example
    static <T> void genericDisplay (T element)
    {
        System.out.println(element.getClass().getName() +
                           " = " + element);
    }

    // Driver method
    public static void main(String[] args)
    {
         // Calling generic method with Integer argument
        genericDisplay(11);

        // Calling generic method with String argument
        genericDisplay("GeeksForGeeks");

        // Calling generic method with double argument
        genericDisplay(1.0);
    }
}
```

Output :

```
java.lang.Integer = 11
java.lang.String = GeeksForGeeks
java.lang.Double = 1.0
```

**Advantages of Generics:**

Programs that uses Generics has got many benefits over non-generic code.

1. Code Reuse: We can write a method/class/interface once and use for any type we want.

.

2. Type Safety : Generics make errors to appear compile time than at run time (It's always better to know problems in your code at compile time rather than making your code fail at run time). Suppose you want to create an ArrayList that store name of students and if by mistake programmer adds an integer object instead of string, compiler allows it. But, when we retrieve this data from ArrayList, it causes problems at runtime.

   ```
   // A Simple Java program to demonstrate that NOT using
   // generics can cause run time exceptions
   import java.util.*;

   class Test
   {
       public static void main(String[] args)
       {
           // Creatinga an ArrayList without any type specified
           ArrayList al = new ArrayList();

           al.add("Sachin");
           al.add("Rahul");
           al.add(10); // Compiler allows this

           String s1 = (String)al.get(0);
           String s2 = (String)al.get(1);

           // Causes Runtime Exception
           String s3 = (String)al.get(2);
       }
   }
   ```

   Output :

   ```
   Exception in thread "main" java.lang.ClassCastException:
      java.lang.Integer cannot be cast to java.lang.String
    at Test.main(Test.java:19)
   ```

   **How generics solve this problem?**

   At the time of defining ArrayList, we can specify that this list can take only String objects.

   ```
   // Using generics converts run time exceptions into
   // compile time exception.
   import java.util.*;
   ```

```
class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Now Compiler doesn't allow this
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);
        String s3 = (String)al.get(2);
    }
}
```

Output:

```
15: error: no suitable method found for add(int)
        al.add(10);
          ^
```

.

3. Individual Type Casting is not needed: If we do not use generics, then, in the above example every-time we retrieve data from ArrayList, we have to typecast it. Typecasting at every retrieval operation is a big headache. If we already know that our list only holds string data then we need not to typecast it every time.

```
// We don't need to typecast individual members of ArrayList
import java.util.*;

class Test
{
    public static void main(String[] args)
    {
        // Creating a an ArrayList with String specified
        ArrayList <String> al = new ArrayList<String> ();

        al.add("Sachin");
        al.add("Rahul");

        // Typecasting is not needed
        String s1 = al.get(0);
        String s2 = al.get(1);
    }
}
```

.

4. Implementing generic algorithms: By using generics, we can implement algorithms that work on different types of objects and at the same they are type safe too.

**References:**
https://docs.oracle.com/javase/tutorial/java/generics/why.html

This article is contributed by **Dharmesh Singh**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

# Implementing our Own Hash Table with Separate Chaining in Java

Every data structure has its own special characteristics for example a BST is used when quick searching of an element (in log(n)) is required. A heap or a priority queue is used when the minimum or maximum element needs to be fetched in constant time. Similarly a hash table is used to fetch, add and remove an element in constant time. It is necessary for anyone to be clear with the working of a hash table before moving on to the implementation aspect. So here is a brief background on the working of hash table and also it should be noted that we will be using Hash Map and Hash Table terminology interchangeably though in Java HashTables are thread safe while HashMaps are not.

The code we are going to implement is available at Link 1 and Link2

But it is strongly recommended that one must read this blog completely and try and decipher the nitty gritty of what goes into implementing a hash map and then try to write the code yourself.

**Background**

Every hash-table stores data in the form of (key, value) combination. Interestingly every key is unique in a Hash Table but values can repeat which means values can be same for different keys present in it. Now as we observe in an array to fetch a value we provide the position/index corresponding to the value in that array. In a Hash Table, instead of an index we use a key to fetch the value corresponding to that key. Now the entire process is described below

Every time a key is generated. The key is passed to a hash function. Every hash function has two parts a **Hash code and a Compressor**.

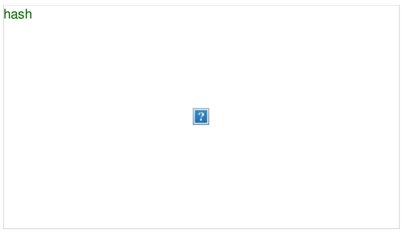
*Hash code is an Integer number* (random or nonrandom). In Java every Object has its own hash code. We will use the hash code generated by JVM in our hash function and to compress the hash code we modulo(%) the hash code by size of the hash table. *So modulo operator is compressor in our implementation.*

The entire process ensures that for any key, we get an integer position within the size of the Hash Table to insert the corresponding value.

So the process is simple, user gives a (key, value) pair set as input and based on the value generated by hash function an index is generated to where the value corresponding to the particular key is stored. So whenever we need to fetch a value corresponding to a key that is just O(1).

This picture stops being so rosy and perfect when the concept of hash collision is introduced. Imagine for different key values same block of hash table is allocated now where do the previously stored values corresponding to some other previous key go. We certainly can't replace it .That will be disastrous! To resolve this issue we will use Separate Chaining Technique, Please note there are other open addressing techniques like double hashing and linear probing whose efficiency is almost same as to that of separate chaining and you can read more about them at Link 1 Link 2 Link3

Now what we do is make a linked list corresponding to the particular bucket of the Hash Table, to accommodate all the values corresponding to different keys who map to the same bucket.



Now there may be a scenario that all the keys get mapped to the same bucket and we have a linked list of n(size of hash table) size from one single bucket, with all the other buckets empty and this is the worst case where a hash table acts a linked list and searching is O(n).So what do we do ?

**Load Factor**

If n be the total number of buckets we decided to fill initially say 10 and let's say 7 of them got filled now, so the load factor is 7/10=0.7.

**In our implementation** whenever we add a key value pair to the Hash Table we check the load factor if it is greater than 0.7 we double the size of our hash table.

**Implementation**

**Hash Node Data Type**
We will try to make a generic map without putting any restrictions on the data type of the key and the value . Also every hash node needs to know the next node it is pointing to in the linked list so a next pointer is also required.

The functions we plan to keep in our hash map are

- **get(K key)** : returns the value corresponding to the key if the key is present in **HT** (**H**ast **T**able)
- **getSize()** : return the size of the HT
- **add()** : adds new valid key, value pair to the HT, if already present updates the value
- **remove()** : removes the key, value pair
- **isEmpty()** : returns true if size is zero

Every Hash Map must have an array list/linked list with an initial size and a bucket size which gets increased by unity every time a key, value pair is added and decreased by unity every time a node is deleted

```
ArrayList<HashNode<K, V>> bucket = new ArrayList<>();
```

A **Helper Function** is implemented to get the index of the key, to avoid redundancy in other functions like get, add and remove. This function uses the in built java function to generate a hash code and we compress the hash code by the size of the HT so that the index is within the range of the size of the HT

**get()**
The get function just takes a key as an input and returns the corresponding value if the key is present in the table otherwise returns null. Steps are:

- Retrieve the input key to find the index in the HT
- Traverse the liked list corresponding to the HT, if you find the value then return it else if you fully traverse the list without returning it means the value is not present in the table and can't be fetched so return null

**remove()**

- Fetch the index corresponding to the input key using the helper function
- The traversal of linked list similar like in get() but what is special here is that one needs to remove the key along with finding it and two cases arise
- If the key to be removed is present at the head of the linked list
- If the key to be removed is not present at head but somewhere else

**add()**
Now to the most interesting and challenging function of this entire implementation.It is interesting because we need to dynamically increase the size of our list when load factor is above the value we specified.

- Just like remove steps till traversal and adding and two cases (addition at head spot or non-head spot) remain the same.
- Towards the end if load factor is greater than 0.7
- We double the size of the array list and then recursively call add function on existing keys because in our case hash value generated uses the size of the array to compress the inbuilt JVM hash code we use ,so we need to fetch new indices for the existing keys. This is very important to understand please re read this paragraph till you get a hang of what is happening in the add function.

Java does in its own implementation of Hash Table uses Binary Search Tree if linked list corresponding to a particular bucket tend to get too long.

```
// Java program to demonstrate implementation of our
// own hash table with chaining for collision detection
import java.util.ArrayList;

// A node of chains
class HashNode<K, V>
{
    K key;
    V value;

    // Reference to next node
    HashNode<K, V> next;

    // Constructor
    public HashNode(K key, V value)
    {
        this.key = key;
        this.value = value;
    }
}

// Class to represent entire hash table
class Map<K, V>
{
    // bucketArray is used to store array of chains
    private ArrayList<HashNode<K, V>> bucketArray;

    // Current capacity of array list
    private int numBuckets;

    // Current size of array list
    private int size;

    // Constructor (Initializes capacity, size and
    // empty chains.
    public Map()
    {
        bucketArray = new ArrayList<>();
        numBuckets = 10;
        size = 0;

        // Create empty chains
        for (int i = 0; i < numBuckets; i++)
```

```
            bucketArray.add(null);
    }

    public int size() { return size; }
    public boolean isEmpty() { return size() == 0; }

    // This implements hash function to find index
    // for a key
    private int getBucketIndex(K key)
    {
        int hashCode = key.hashCode();
        int index = hashCode % numBuckets;
        return index;
    }

    // Method to remove a given key
    public V remove(K key)
    {
        // Apply hash function to find index for given key
        int bucketIndex = getBucketIndex(key);

        // Get head of chain
        HashNode<K, V> head = bucketArray.get(bucketIndex);

        // Search for key in its chain
        HashNode<K, V> prev = null;
        while (head != null)
        {
            // If Key found
            if (head.key.equals(key))
                break;

            // Else keep moving in chain
            prev = head;
            head = head.next;
        }

        // If key was not there
        if (head == null)
            return null;

        // Reduce size
        size--;

        // Remove key
        if (prev != null)
            prev.next = head.next;
        else
            bucketArray.set(bucketIndex, head.next);

        return head.value;
    }

    // Returns value for a key
    public V get(K key)
    {
        // Find head of chain for given key
        int bucketIndex = getBucketIndex(key);
        HashNode<K, V> head = bucketArray.get(bucketIndex);

        // Search key in chain
        while (head != null)
        {
            if (head.key.equals(key))
                return head.value;
            head = head.next;
        }

        // If key not found
        return null;
    }

    // Adds a key value pair to hash
    public void add(K key, V value)
    {
        // Find head of chain for given key
        int bucketIndex = getBucketIndex(key);
        HashNode<K, V> head = bucketArray.get(bucketIndex);

        // Check if key is already present
        while (head != null)
        {
            if (head.key.equals(key))
            {
                head.value = value;
                return;
            }
            head = head.next;
        }

        // Insert key in chain
        size++;
        head = bucketArray.get(bucketIndex);
        HashNode<K, V> newNode = new HashNode<K, V>(key, value);
        newNode.next = head;
        bucketArray.set(bucketIndex, newNode);

        // If load factor goes beyond threshold, then
        // double hash table size
        if ((1.0*size)/numBuckets >= 0.7)
        {
            ArrayList<HashNode<K, V>> temp = bucketArray;
            bucketArray = new ArrayList<>();
            numBuckets = 2 * numBuckets;
            size = 0;
            for (int i = 0; i < numBuckets; i++)
                bucketArray.add(null);

            for (HashNode<K, V> headNode : temp)
            {
                while (headNode != null)
                {
                    add(headNode.key, headNode.value);
                    headNode = headNode.next;
                }
            }
        }
    }

    // Driver method to test Map class
    public static void main(String[] args)
    {
    Map<String, Integer>map = new Map<>();
    map.add("this",1 );
    map.add("coder",2 );
    map.add("this",4 );
    map.add("hi",5 );
        System.out.println(map.size());
        System.out.println(map.remove("this"));
        System.out.println(map.remove("this"));
        System.out.println(map.size());
        System.out.println(map.isEmpty());
    }
    }
```

Output :

```
3
4
null
2
false
```

The entire code is available at https://github.com/ishaan007/Data-Structures/blob/master/HashMaps/Map.java

This article is contributed by **Ishaan Arora.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Corner    Company Wise Coding Practice

Hash
Java

# Quick ways to check for Prime and find next Prime in Java

Many programming contest problems are somehow related Prime Numbers. Either we are required to check Prime Numbers, or we are asked to perform certain functions for all prime number between 1 to N. Example: Calculate the sum of all prime numbers between 1 and 1000000.

Java provides two function under java.math.BigInteger to deal with Prime Numbers.

- **isProbablePrime(int certainty):** A method in BigInteger class to check if a given number is prime. For certainty = 1, it return true if BigInteger is prime and false if BigInteger is composite.
  Below is Java program to demonstrate above function.

```
// A Java program to check if a number is prime using
// inbuilt function
import java.util.*;
import java.math.*;

class CheckPrimeTest
{
    //Function to check and return prime numbers
    static boolean checkPrime(long n)
    {
        // Converting long to BigInteger
        BigInteger b = new BigInteger(String.valueOf(n));

        return b.isProbablePrime(1);
    }

    // Driver method
    public static void main (String[] args)
                     throws java.lang.Exception
    {
       long n = 13;
       System.out.println(checkPrime(n));
    }
}
```

  Output:

```
true
```

- **nextProbablePrime()** : Another method present in BigInteger class. This functions returns the next Prime Number greater than current BigInteger.
  Below is Java program to demonstrate above function.

```
// Java program to find prime number greater than a
// given number using built in method
import java.util.*;
import java.math.*;

class NextPrimeTest
{
    // Function to get nextPrimeNumber
    static long nextPrime(long n)
    {
        BigInteger b = new BigInteger(String.valueOf(n));
        return Long.parseLong(b.nextProbablePrime().toString());
    }

    // Driver method
    public static void main (String[] args)
                   throws java.lang.Exception
    {
        long n = 14;
        System.out.println(nextPrime(n));
    }
}
```

Output:

```
17
```

This article is contributed by **Ayush Jain**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Competitive Programming
Java
prime-number

# Static vs Dynamic Binding in Java

**Static Binding:** The binding which can be resolved at compile time by compiler is known as static or early binding. Binding of all the static, private and final methods is done at compile-time .

**Why binding of static, final and private methods is always a static binding?**
Static binding is better performance wise (no extra overhead is required). Compiler knows that all such methods **cannot be overridden** and will always be accessed by object of local class. Hence compiler doesn't have any difficulty to determine object of class (local class for sure). That's the reason binding for such methods is static.
Let's see by an example

```
public class NewClass
{
    public static class superclass
    {
        static void print()
        {
            System.out.println("print in superclass.");
        }
    }
    public static class subclass extends superclass
    {
        static void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

Before scrolling further down, Guess the output of the above program?

**Output**:

```
print in superclass.
print in superclass.
```

As you can see, in both cases print method of superclass is called. Lets see how this happens

- We have created one object of subclass and one object of superclass with the reference of the superclass.
- Since the print method of superclass is static, compiler knows that it will not be overridden in subclasses and hence compiler knows during compile time which print method to call and hence no ambiguity.

*As an exercise, reader can change the reference of object B to subclass and then check the output.*

**Dynamic Binding:** In Dynamic binding compiler doesn't decide the method to be called. Overriding is a perfect example of dynamic binding. In overriding both parent and child classes have same method . Let's see by an example

```
public class NewClass
{
    public static class superclass
    {
        void print()
        {
            System.out.println("print in superclass.");
        }
    }

    public static class subclass extends superclass
    {
        @Override
        void print()
        {
            System.out.println("print in subclass.");
        }
    }

    public static void main(String[] args)
    {
        superclass A = new superclass();
        superclass B = new subclass();
        A.print();
        B.print();
    }
}
```

**Output:**

```
print in superclass.
print in subclass.
```

Here the output differs. But why? Let's break down the code and understand it thoroughly.

- Methods are not static in this code.
- During compilation, the compiler has no idea as to which print has to be called since compiler goes only by referencing variable not by type of object and therefore the binding would be delayed to runtime and

therefore the corresponding version of print will be called based on type on object.

**Important Points**

- private, final and static members (methods and variables) use static binding while for virtual methods (In Java methods are virtual by default) binding is done during run time based upon run time object.
- Static binding uses Type information for binding while Dynamic binding uses Objects to resolve binding.
- Overloaded methods are resolved (deciding which method to be called when there are multiple methods with same name) using static binding while overridden methods using dynamic binding, i.e, at run time.

This article is contributed by **Rishabh Mahrsee.** If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**GATE CS Corner    Company Wise Coding Practice**

Java

## Command Line arguments in Java

If we run a Java Program by writing the command "**java Hello Geeks At GeeksForGeeks**" where the name of the class is "Hello", then it will run upto Hello. It is command upto "Hello" and after that i.e "Geeks At GeeksForGeeks", these are command line arguments.

When command line arguments are supplied to JVM, JVM wraps these and supply to args[]. It can be confirmed that they are actually wrapped up in args array by checking the length of args using args.length

```
// Program to check for command line arguments
class Hello
{
    public static void main(String[] args)
    {
        // check if length of args array is
        // greater than 0
        if (args.length > 0)
        {
            System.out.println("The command line"+
                               " arguments are:");

            // iterating the args array and printing
            // the command line arguments
            for (String val:args)
                System.out.println(val);
        }
        else
            System.out.println("No command line "+
                               "arguments found.");
    }
}
```

**When executed as following:**
java Hello Geeks at GeeksforGeeks
Output:

```
The command line arguments are:
Geeks
at
GeeksforGeeks
```

**When executed as following:**
java Hello
Output:

```
No command line arguments found.
```

This article is contributed by **Twinkle Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java

## How to create Immutable class in Java?

Immutable class means that once an object is created, we cannot change its content. In Java, all the wrapper classes (like String, Boolean, Byte, Short) and String class is immutable. We can create our own immutable class as well.

Following are the requirements:
• Class must be declared as final (So that child classes can't be created)
• Data members in the class must be declared as final (So that we can't change the value of it after object creation)
• A parameterized constructor
• Getter method for all the variables in it
• No setters(To not have option to change the value of the instance variable)

**Example to create Immutable class**

```
// An immutable class
public final class Student
{
    final String name;
    final int regNo;

    public Student(String name,int regNo)
    {
        this.name=name;
        this.regNo=regNo;
    }
    public String getName()
    {
        return name;
    }
    public int getRegNo()
    {
        return regNo;
    }
}

// Driver class
class Test
{
    public static void main(String args[])
    {
        Student s = new Student("ABC", 101);
        System.out.println(s.name);
        System.out.println(s.regNo);

        // Uncommenting below line causes error
        // s.regNo = 102;
    }
}
```

Output :

```
ABC
101
```

In this example, we have created a final class named Student. It has two final data members, a parameterized constructor and getter methods. Please note that there is no setter method here.
(In order to make it functional, create objects of Student class in the main function.)

This article is contributed by **Abhishree Shetty**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Java instanceof and its applications

instanceof is a keyword that is used for checking if a reference variable is containing a given type of object reference or not.

Following is a Java program to show different behaviors of instanceof.

```
// Java program to demonstrate working of instanceof

// Creating sample classes with parent Child
// relationship
class Parent {  }
class Child extends Parent { }

class Test
{
    public static void main(String[] args)
    {
        Child cobj = new Child();

        // A simple case
        if (cobj instanceof Child)
           System.out.println("cobj is instance of Child");
        else
           System.out.println("cobj is NOT instance of Child");

        // instanceof returns true for Parent class also
        if (cobj instanceof Parent)
           System.out.println("cobj is instance of Parent");
        else
           System.out.println("cobj is NOT instance of Parent");

        // instanceof returns true for all ancestors (Note : Object
        // is ancestor of all classes in Java)
        if (cobj instanceof Object)
           System.out.println("cobj is instance of Object");
        else
           System.out.println("cobj is NOT instance of Object");
    }
}
```

Output :

```
cobj is instance of Child
cobj is instance of Parent
cobj is instance of Object
```

**instanceof returns false for null**

```
// Java program to demonstrate that instanceof
// returns false for null

class Test {  }

class Main
{
    public static void main(String[] args)
    {
        Test tobj = null

        // A simple case
        if (tobj instanceof Test)
           System.out.println("tobj is instance of Test");
        else
           System.out.println("tobj is NOT instance of Test");
    }
}
```

Output :

```
tobj is NOT instance of Test
```

**A parent object is not an instance of Child**

```
// A Java program to show that a parent object is
// not an instance of Child

class Parent {  }
class Child extends Parent { }

class Test
{
    public static void main(String[] args)
    {
        Parent pobj = new Parent();
        if (pobj instanceof Child)
           System.out.println("pobj is instance of Child");
        else
           System.out.println("pobj is NOT instance of Child");
    }
}
```

Output :

```
tobj is NOT instance of Child
```

**A parent reference referring to a Child is an instance of Child**

```
// A Java program to show that a parent reference
// referring to a Child is an instance of Child

class Parent {  }
class Child extends Parent { }

class Test
{
    public static void main(String[] args)
    {
        // Reference is Parent type but object is
        // of child type.
        Parent cobj = new Child();
        if (cobj instanceof Child)
           System.out.println("cobj is instance of Child");
        else
           System.out.println("cobj is NOT instance of Child");
    }
}
```

Output :

```
tobj is an instance of Child
```

**Application:**

We have seen here that a parent class data member is accessed when a reference of parent type refers to a child object. We can access child data member using type casting.
Syntax: (child_class_name) Parent_Reference_variable).func.name().

When we do typecasting, it is always a good idea to check if the typecasting is valid or not. instanceof helps use here. We can always first check for validity using instancef, then do typecasting.

```
// A Java program to demonstrate that non-method
// members are accessed according to reference
// type (Unlike methods which are accessed according
// to the referred object)

class Parent
{
    int value = 1000;
}

class Child extends Parent
{
    int value = 10;
}

// Driver class
class Test
{
    public static void main(String[] args)
    {
        Parent cobj = new Child();
        Parent par = cobj;

        // Using instanceof to make sure that par
        // is a valid reference before typecasting
        if (par instanceof Child)
        {
            System.out.println("Value accessed through " +
                "parent reference with typecasting is " +
                                    ((Child)par).value);
        }
    }
}
```

Output:

```
Value accessed through parent reference with typecasting is 10
```

This article is contributed by **Twinkle Tyagi** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Date class in Java (With Examples)

The class Date represents a specific instant in time, with millisecond precision. The Date class of java.util package implements Serializable, Cloneable and Comparable interface. It provides constructors and methods to deal with date and time with java.

**Constructors**

- **Date()** : Creates date object representing current date and time.
- **Date(long milliseconds)** : Creates a date object for the given milliseconds since January 1, 1970, 00:00:00 GMT.
- **Date(int year, int month, int date)**
- **Date(int year, int month, int date, int hrs, int min)**
- **Date(int year, int month, int date, int hrs, int min, int sec)**
- **Date(String s)**

**Note :** The last 4 constructors of the Date class are Deprecated.

```
// Java program to demonstrate constuctors of Date
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        Date d1 = new Date();
        System.out.println("Current date is " + d1);
        Date d2 = new Date(2323223232L);
        System.out.println("Date represented is "+ d2 );
    }
}
```

Output:

```
Current date is Tue Jul 12 18:35:37 IST 2016
Date represented is Wed Jan 28 02:50:23 IST 1970
```

**Important Methods**

- **boolean after(Date date) :** Tests if current date is after the given date.
- **boolean before(Date date) :** Tests if current date is before the given date.
- **int compareTo(Date date) :** Compares current date with given date. Returns 0 if the argument Date is equal to the Date; a value less than 0 if the Date is before the Date argument; and a value greater than 0 if the Date is after the Date argument.
- **long getTime()** : Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
- **void setTime(long time)** : Changes the current date and time to given time.

```
// Program to demonstrate methods of Date class
import java.util.*;

public class Main
{
    public static void main(String[] args)
    {
        // Creating date
        Date d1 = new Date(2000, 11, 21);
        Date d2 = new Date();  // Current date
        Date d3 = new Date(2010, 1, 3);

        boolean a = d3.after(d1);
        System.out.println("Date d3 comes after " +
                   "date d2: " + a);

        boolean b = d3.before(d2);
        System.out.println("Date d3 comes before "+
                   "date d2: " + b);

        int c = d1.compareTo(d2);
        System.out.println(c);

        System.out.println("Miliseconds from Jan 1 "+
              "1970 to date d1 is " + d1.getTime());

        System.out.println("Before setting "+d2);
        d2.setTime(204587433443L);
        System.out.println("After setting "+d2);
    }
}
```

Output:

```
Date d3 comes after date d2: true
Date d3 comes before date d2: false
1
Miliseconds from Jan 1 1970 to date d1 is 60935500800000
Before setting Tue Jul 12 13:13:16 UTC 2016
After setting Fri Jun 25 21:50:33 UTC 1976
```

References: http://www.javatpoint.com/java-util-date

This article is contributed by **Rahul Agrawal** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# System.exit() in Java

The **java.lang.System.exit()** method exits current program by terminating running Java virtual machine. This method takes a status code. A non-zero value of status code is generally used to indicate abnormal termination. This is similar exit in C/C++.

Following is the declaration for **java.lang.System.exit()** method:

```
public static void exit(int status)
```

**exit(0)** : Generally used to indicate successful termination.
**exit(1) or exit(-1) or any other non-zero value** – Generally indicates unsuccessful termination.

**Note :** This method does not return any value.

The following example shows the usage of **java.lang.System.exit()** method.

```
// A Java program to demonstrate working of exit()
import java.util.*;
import java.lang.*;

class GfG
{
    public static void main(String[] args)
    {
        int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};

        for (int i = 0; i < arr.length; i++)
        {
            if (arr[i] >= 5)
            {
                System.out.println("exit...");

                // Terminate JVM
                System.exit(0);
            }
            else
                System.out.println("arr["+i+"] = " +
                                  arr[i]);
        }
        System.out.println("End of Program");
    }
}
```

Output:

```
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
exit...
```

**Reference :**
https://docs.oracle.com/javase/7/docs/api/java/lang/Runtime.html

This article is contributed by **Amit Khandelwal** .If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Generating random numbers in Java

Java provides three ways to generate random numbers using some built-in methods and classes as listed below:

- **java.util.Random** class
- **Math.random** method : Can Generate Random Numbers of double type.
- **ThreadLocalRandom** class

**1) java.util.Random**

- For using this class to generate random numbers, we have to first create an instance of this class and then invoke methods such as nextInt(), nextDouble(), nextLong() etc using that instance.
- We can generate random numbers of types integers, float, double, long, booleans using this class.
- We can pass arguments to the methods for placing an upper bound on the range of the numbers to be generated. For example, nextInt(6) will generate numbers in the range 0 to 5 both inclusive.

```
// A Java program to demonstrate random number generation
// using java.util.Random;
import java.util.Random;

public class generateRandom{

    public static void main(String args[])
    {
        // create instance of Random class
        Random rand = new Random();

        // Generate random integers in range 0 to 999
        int rand_int1 = rand.nextInt(1000);
        int rand_int2 = rand.nextInt(1000);

        // Print random integers
        System.out.println("Random Integers: "+rand_int1);
        System.out.println("Random Integers: "+rand_int2);

        // Generate Random doubles
        double rand_dub1 = rand.nextDouble();
        double rand_dub2 = rand.nextDouble();

        // Print random doubles
        System.out.println("Random Doubles: "+rand_dub1);
        System.out.println("Random Doubles: "+rand_dub2);
    }
}
```

Output:

```
Random Integers: 547
Random Integers: 126
Random Doubles: 0.8369779739988428
Random Doubles: 0.5497554388209912
```

**2) Math.random()**

The class Math contains various methods for performing various numeric operations such as, calculating exponentiation, logarithms etc. One of these methods is random(), this method returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. The returned values are chosen pseudorandomly. This method can only generate random numbers of type Doubles. Below program explains how to use this method:

```
// Java program to demonstrate working of
// Math.random() to generate random numbers
```

```
import java.util.*;

public class generateRandom
{
    public static void main(String args[])
    {
        // Generating random doubles
        System.out.println("Random doubles: " + Math.random());
        System.out.println("Random doubles: " + Math.random());
    }
}
```

Output:

```
Random doubles: 0.13077348615666562
Random doubles: 0.09247016928442775
```

**3) java.util.concurrent.ThreadLocalRandom class**

This class is introduced in java 1.7 to generate random numbers of type integers, doubles, booleans etc. Below program explains how to use this class to generate random numbers:

```
// Java program to demonstrate working of ThreadLocalRandom
// to generate random numbers.
import java.util.concurrent.ThreadLocalRandom;

public class generateRandom
{
    public static void main(String args[])
    {
        // Generate random integers in range 0 to 999
        int rand_int1 = ThreadLocalRandom.current().nextInt();
        int rand_int2 = ThreadLocalRandom.current().nextInt();

        // Print random integers
        System.out.println("Random Integers: " + rand_int1);
        System.out.println("Random Integers: " + rand_int2);

        // Generate Random doubles
        double rand_dub1 = ThreadLocalRandom.current().nextDouble();
        double rand_dub2 = ThreadLocalRandom.current().nextDouble();

        // Print random doubles
        System.out.println("Random Doubles: " + rand_dub1);
        System.out.println("Random Doubles: " + rand_dub2);

        // Generate random booleans
        boolean rand_bool1 = ThreadLocalRandom.current().nextBoolean();
        boolean rand_bool2 = ThreadLocalRandom.current().nextBoolean();

        // Print random Booleans
        System.out.println("Random Booleans: " + rand_bool1);
        System.out.println("Random Booleans: " + rand_bool2);
    }
}
```

Output:

```
Random Integers: 536953314
Random Integers: 25905330
Random Doubles: 0.7504989954390163
Random Doubles: 0.7658597196204409
Random Booleans: false
Random Booleans: true
```

**References:**

https://docs.oracle.com

This article is contributed by **Harsh Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# BitSet class in Java with Examples | Set 1

**BitSet** is a class defined in the java.util package. It creates an array of bits represented by boolean values.

**Constructors:**

```
BitSet class Constructors
  /              \
BitSet()        BitSet(int no_Of_Bits)
```

- **BitSet() :** A no-argument constructor to create an empty BitSet object.
- **BitSet(int no_Of_Bits) :** A one-constructor with an integer argument to create an instance of the BitSet class with an initial size of the integer argument representing the number of bits.

```
// Java program illustrating Bitset Class constructors.
import java.util.*;
public class GFG
{
    public static void main(String[] args)
    {
        // Constructors of BitSet class
        BitSet bs1 = new BitSet();
        BitSet bs2 = new BitSet(6);

        /* set is BitSet class method
           expalined in next articles */
        bs1.set(0);
        bs1.set(1);
        bs1.set(2);
        bs1.set(4);
        // assign values to bs2
        bs2.set(4);
        bs2.set(6);
        bs2.set(5);
        bs2.set(1);
        bs2.set(2);
        bs2.set(3);

        // Printing the 2 Bitsets
        System.out.println("bs1  : " + bs1);
        System.out.println("bs2  : " + bs2);
    }
}
```

Output:

```
bs1 : {0, 1, 2, 4}
bs2 : {1, 2, 3, 4, 5, 6}
```

**Important Points :**

- The size of the array is flexible and can grow to accommodate additional bit as needed.
- As it is an array, the index is zero-based and the bit values can be accessed only by non-negative integers as an index .
- The default value of the BitSet is boolean false with a representation as 0 (off).
- Calling the clear method makes the bit values set to false.
- BitSet uses about 1 bit per boolean value.
- To access a specific value in the BitSet, the get method is used with an integer argument as an index.

**What happens if the array index in bitset is set as Negative?**

The program will throw java.lang.NegativeArraySizeException

```
// Java program illustrating Exception when we access
// out of index in BitSet class.
import java.util.*;

public class GFG
{
    public static void main(String[] args)
    {
        // Constructors of BitSet class
        BitSet bs1 = new BitSet();

        // Negative array size
        BitSet bs2 = new BitSet(-1);

        /* set is BitSet class method
           expalined in next articles */
        // assigning values to bitset 1
        bs1.set(0);
        bs1.set(1);

        // assign values to bs2
        bs2.set(4);
        bs2.set(6);

        System.out.println("bs1  : " + bs1);
        System.out.println("bs2  : " + bs2);
    }
}
```

Output:

```
Exception in thread "main" java.lang.NegativeArraySizeException: nbits < 0: -1
 at java.util.BitSet.(BitSet.java:159)
 at GFG.main(NewClass.java:9)
```

BitSet class methods in Java with Examples | Set 2

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Library

# BitSet class methods in Java with Examples | Set 2

```
Methods discussed in this post:
                    BitSet class methods.
             /     /    |     |      \      \
         set()  xor()  clone() clear()  length()  cardinality()
```

We strongly recommend to refer below set 1 as a prerequisite of this.
**BitSet class in Java | Set 1**

1. **set() : java.util.BitSet.set()** method is a sets the bit at the specified index to the specified value.
   **Syntax:**

   ```
   public void set(int bitpos)
   public void set(int bitpos, boolean val)
   Parameters:
   bitpos : a bit index
   val : a boolean value to set
   Return: Nothing
   Throws: IndexOutOfBoundsException - if the specified index is negative
   ```

2. **clone() : java.util.BitSet.clone()** method clones a BitSet produces a new BitSet that is equal to it. The clone of the bit set is another bit set that has exactly the same bits set to true as this bit set.
   **Syntax:**

   ```
   public Object clone()
   Return: a clone of this bit set
   ```

3. **cardinality : java.util.BitSet.cardinality()** method is used to find the no. of elements in Bitset.

**Syntax:**

```
public int cardinality()
Return: the number of bits set to true in this BitSet.
```

```
// Java program explaining BitSet class methods
// set(), clone(), cardinality()
import java.util.*;
public class NewClasss
{
    public static void main(String[] args)
    {
        BitSet bs1 = new BitSet();
        BitSet bs2 = new BitSet(8);
        BitSet bs3 = new BitSet();

        // assign values to bs1 using set()
        bs1.set(0);
        bs1.set(1);
        bs1.set(2);
        bs1.set(4);

        // assign values to bs2
        bs2.set(4);
        bs2.set(6);
        bs2.set(5);

        // Here we are using .clone() method to make
        // bs3 as bs1
        bs3 = (BitSet) bs1.clone();

        // Printing the 3 Bitsets
        System.out.println("bs1 : " + bs1);
        System.out.println("bs2 : " + bs2);
        System.out.println("bs3 cloned from bs1 : " + bs3);

        // Using .cardinality() method to print the no. of
        // elements of Bitset
        System.out.println("Cardinality of bs1 : " +
                                   bs1.cardinality());
        System.out.println("Cardinality of bs2 : " +
                                   bs2.cardinality());
    }
}
```

Output:

```
bs1 : {0, 1, 2, 4}
bs2 : {4, 5, 6}
bs3 cloned from bs1 : {0, 1, 2, 4}
Cardinality of bs1 : 4
Cardinality of bs2 : 3
```

4. **clear() : java.util.BitSet.clear()** method is used to clear the elements i.e. set all the Bitset elements to false.
   **Syntax:**

```
public void clear(int frompos,int topos)
public void clear(int bitIndex)
public void clear()
Parameters:
frompos - index of the first bit to be cleared
topos - index after the last bit to be cleared
bitIndex - the index of the bit to be cleared
Throws:
IndexOutOfBoundsException - if pos value is negative or frompos is larger than topos
```

5. **xor() : java.util.BitSet.xor()** method performs the logical Xor operation on the bitsets.
   This bit set is modified so that a bit in it has the value true if and only if :
   - The bit initially has the value true, and the corresponding bit in the argument has the value false.
   - The bit initially has the value false, and the corresponding bit in the argument has the value true.

   **Syntax:**

```
public void xor(BitSet set)
Parameters:
set - the BitSet with which we need to perform operation
```

6. **length() : java.util.BitSet.length()** method return returns logical size of the bitset.
   The index of the highest set bit in the bitset plus one. Returns zero if the bitset contains no set bits.
   **Syntax:**

```
public int length()
Returns:
the logical size of this BitSet
```

```
// Java program explaining BitSet class methods
//  xor(), length(), clear() methods
import java.util.*;
public class NewClass
{
    public static void main(String[] args)
    {
        BitSet bs1 = new BitSet();
        BitSet bs2 = new BitSet();

        // assign values to bs1 using set()
        bs1.set(7);
        bs1.set(1);
        bs1.set(2);
        bs1.set(4);
        bs1.set(3);
        bs1.set(6);

        // assign values to bs2
        bs2.set(4);
        bs2.set(6);
        bs2.set(3);
        bs2.set(9);
        bs2.set(2);
        // Printing the Bitsets
        System.out.println("bs1 : " + bs1);
        System.out.println("bs2 : " + bs2);

        // use of length() method
        System.out.println("use of length() : " + bs1.length());

        // use of xor() to perform logical Xor operation
        bs1.xor(bs2);
        System.out.println("Use of xor() : " + bs1);
        bs2.xor(bs1);
        System.out.println("Use of xor() : " + bs2);

        // clear from index 2 to index 4 in bs1
        bs2.clear(1, 2);
        System.out.println("bs2 after clear method : " + bs2);

        // clear complete Bitset
        bs1.clear();
        System.out.println("bs1 after clear method : " + bs1);
    }
}
```

Output:

```
bs1 : {1, 2, 3, 4, 6, 7}
bs2 : {2, 3, 4, 6, 9}
use of length() : 8
Use of xor() : {1, 7, 9}
Use of xor() : {1, 2, 3, 4, 6, 7}
bs2 after clear method : {2, 3, 4, 6, 7}
bs1 after clear method : {}
```

**BitSet class methods in Java with Examples | Set 3**

**References :**
https://docs.oracle.com/javase/7/docs/api/java/util/BitSet.html

This article is contributed by **Mohit Gupta**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Java-Library

# NaN (Not a Number) in Java

Can You guess the output of following code fragment:

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(2 % 0);
    }
}
```

Yes, You guessed it right: ArithmeticException
Output:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
 at traps.Test.main(Test.java:3)
```

Now guess the Output of :

```
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(2.0 % 0);
    }
}
```

Did you guessed it right ?
Output:

```
NaN
```

**What is NaN?**

"**NaN**" stands for "not a number". "Nan" is produced if a floating point operation has some input parameters that cause the operation to produce some undefined result. For example, **0.0 divided by 0.0** is arithmetically undefined. Finding out the **square root of a negative number** too is undefined.

```
//Java Program to illustrate NaN
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(2.0 % 0);
        System.out.println(0.0 / 0);
        System.out.println(Math.sqrt(-1));
    }
}
```

Output:

```
NaN
NaN
NaN
```

In javadoc, the constant field NaN is declared as following in the Float and Double Classes respectively.

```
public static final float  NaN = 0f / 0f;
public static final double      NaN = 0d / 0d;
```

**How to Compare NaN Values?**

All numeric operations with NaN as an operand produce NaN as a result. Reason behind this is that NaN is unordered, so a numeric comparison operation involving one or two NaNs returns false.

- The numerical comparison operators , and >= always return false if either or both operands are NaN.(§15.20.1)
- The equality operator == returns false if either operand is NaN.
- The inequality operator != returns true if either operand is NaN . (§15.21.1)

```
// Java program to test relational operator on NaN
public class ComparingNaN
{
 public static void main(String[] args)
 {
  // comparing NaN constant field defined in
  // Float Class
  System.out.print("Check if equal :");
  System.out.println(Float.NaN == Float.NaN);

  System.out.print("Check if UNequal: ");
  System.out.println(Float.NaN != Float.NaN);

  // comparing NaN constant field defined in Double Class
  System.out.print("Check if equal: ");
  System.out.println(Double.NaN == Double.NaN);

  System.out.print("Check if UNequal: ");
  System.out.println(Double.NaN != Double.NaN);


  // More Examples
  double NaN = 2.1 % 0;
  System.out.println((2.1%0) == NaN);
  System.out.println(NaN == NaN);
 }
}
```

Output:

```
Check if equal :false
Check if UNequal: true
Check if equal: false
Check if UNequal: true
false
false
```

**isNaN() method**

This method returns true if the value represented by this object is NaN; false otherwise.

```

import java.lang.*;

public class isNan
{

   public static void main(String[] args)
   {

     Double x = new Double(-2.0/0.0);
     Double y = new Double(0.0/0.0);


     // returns false if this Double value is not a Not-a-Number (NaN)
     System.out.println(y + " = " + y.isNaN());

     // returns true if this Double value is a Not-a-Number (NaN)
     System.out.println(x + " = " + x.isNaN());

   }
}
```

Output:

```
NaN = true
-Infinity = false
```

**Floating type doesn't produces Exception while operating with mathematical values**

IEEE 754 floating point numbers can represent positive or negative infinity, and NaN (not a number). These three values arise from calculations whose result is undefined or cannot be represented accurately.
Java is following known math facts. 1.0 / 0.0 is infinity, but the others are indeterminate forms, which Java represents as NaN (not a number).

```
// Java program to illustrate output of floating
// point number operations
public class Test
{
    public static void main(String[] args)
    {
        System.out.println(2.0 / 0);
        System.out.println(-2.0 / 0);
        System.out.println(9.0E234 / 0.1E-234);
    }
}
```

Output:

```
Infinity
-Infinity
Infinity
```

References:
https://docs.oracle.com/javase/7/docs/api/java/lang/Double.html
https://docs.oracle.com/javase/specs/jls/se7/html/jls-4.html

This article is contributed by **Pankaj kumar**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Generating Password and OTP in Java

You may go through Generate a One Time Password or Unique Identification URL article before this for better understanding.

Generating Password and OTP in Java

Many a times we forget our passwords and we opt for Forget password option and within no time we get a new password at our registered email-ID or phone no. to login our account. And every time we get a different password.
Sometime we access our bank accounts while shopping from an online store or many more ways, in order to verify our transition from the bank account they send us OTP(One Time Password) on our registered phone no. or our email-ID, within no time.

The following code explains how to generate such Passwords and OTP within no time and what code we can use if in case we need to do so.

**Java program explaining the generation of Password**

```
// Java code to explain how to generate random
// password

// Here we are using random() method of util
// class in Java
import java.util.*;

public class NewClass
{
    public static void main(String[] args)
    {
        // Length of your password as I have choose
        // here to be 8
        int length = 10;
        System.out.println(geek_Password(length));
    }

    // This our Password generating method
    // We have use static here, so that we not to
    // make any object for it
    static char[] geek_Password(int len)
    {
        System.out.println("Generating password using random() : ");
        System.out.print("Your new password is : ");

        // A strong password has Cap_chars, Lower_chars,
        // numeric value and symbols. So we are using all of
        // them to generate our password
        String Capital_chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
        String Small_chars = "abcdefghijklmnopqrstuvwxyz";
        String numbers = "0123456789";
                String symbols = "!@#$%^&*_=+-/.?<>)";


        String values = Capital_chars + Small_chars +
                        numbers + symbols;

        // Using random method
        Random rndm_method = new Random();

        char[] password = new char[len];

        for (int i = 0; i < len; i++)
        {
            // Use of charAt() method : to get character value
            // Use of nextInt() as it is scanning the value as int
            password[i] =
              values.charAt(rndm_method.nextInt(values.length()));

        }
        return password;
    }
}
```

**Note :** The password we are generating will change every time. As we have used random() method to generate the password.
**Output :**

```
Generating password using random() :
Your new password is : KHeCZBTM;-
```

**Java program explaining the generation of OTP(One Time Password)**

```
// Java code to explain how to generate OTP

// Here we are using random() method of util
// class in Java
import java.util.*;
```

```
public class NewClass
{
    static char[] OTP(int len)
    {
        System.out.println("Generating OTP using random() : ");
        System.out.print("You OTP is : ");

        // Using numeric values
        String numbers = "0123456789";

        // Using random method
        Random rndm_method = new Random();

        char[] otp = new char[len];

        for (int i = 0; i < len; i++)
        {
            // Use of charAt() method : to get character value
            // Use of nextInt() as it is scanning the value as int
            otp[i] =
             numbers.charAt(rndm_method.nextInt(numbers.length()));
        }
        return otp;
    }
    public static void main(String[] args)
    {
        int length = 4;
        System.out.println(OTP(length));
    }
}
```

**Note :**
The OTP we are generating will change every time. As we have used random() method to generate the OTP.
**Output :**

```
Generating OTP using random() :
You OTP is : 5291
```

This article is contributed by **Mohit Gupta** . If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

GBlog
Java
Project
TechTips

## Why to use char[] array over a string for storing passwords in Java?

1. **Strings are immutable:** Strings are immutable in Java and therefore if a password is stored as plain text it will be available in memory until Garbage collector clears it and as Strings are used in String pool for re-usability there are high chances that it will remain in memory for long duration, which is a security threat. Strings are immutable and there is no way that the content of Strings can be changed because any change will produce new String.
   With an array, the data can be wiped explicitly data after its work is complete. The array can be overwritten and and the password won't be present anywhere in the system, even before garbage collection.
2. **Security:** Any one who has access to memory dump can find the password in clear text and that's another reason to use encrypted password than plain text. So Storing password in character array clearly mitigates security risk of stealing password.
3. **Log file safety:** With an array, one can explicitly wipe the data , overwrite the array and the password won't be present anywhere in the system.
   With plain String, there are much higher chances of accidentally printing the password to logs, monitors or some other insecure place. char[] is less vulnerable.

```
//Java program to illustate prefering char[] arrays
//over strings for passwords in Java
public class PasswordPreference
{

    public static void main(String[] args)
    {

        String strPwd = "password";
        char[] charPwd = new char[] {'p','a','s','s','w','o','r','d'};

        System.out.println("String password: " + strPwd );
        System.out.println("Character password: " + charPwd );
    }
}
```

   Output:

```
String password: password
Character password: [C@15db9742
```

4. **Java Recommendation:** Java has methods like JPasswordField of javax.swing as the method public String getText() which returns String is Deprecated from Java 2 and is replaced by public char[] getPassword() which returns Char Array.

This article is contributed by **Kanika Tyagi**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**GATE CS Corner    Company Wise Coding Practice**

Java

## Programs for printing pyramid patterns in Java

This article is aimed at giving a Java implementation for pattern printing.

- **Simple pyramid pattern**

```
import java.io.*;

// Java code to demonstrate star patterns
public class GeeksForGeeks
{
    // Function to demonstrate printing pattern
    public static void printStars(int n)
    {
        int i, j;

        // outer loop to handle number of rows
        //  n in this case
        for(i=0; i<n; i++)
        {

            //  inner loop to handle number of columns
            //  values changing acc. to outer loop
            for(j=0; j<=i; j++)
            {
                // printing stars
                System.out.print("* ");
            }

            // ending line after each row
            System.out.println();
        }
    }

    // Driver Function
    public static void main(String args[])
    {
        int n = 5;
        printStars(n);
    }
}
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

- **After 180 degree rotation**

```
import java.io.*;

// Java code to demonstrate star pattern
public class GeeksForGeeks
{
    // Function to demonstrate printing pattern
    public static void printStars(int n)
    {
        // number of spaces
        int i, j, k=2*n-2;

        // outer loop to handle number of rows
        //  n in this case
        for(i=0; i<n; i++)
        {

            // inner loop to handle number spaces
            // values changing acc. to requirement
            for(j=0; j<k; j++)
            {
                // printing spaces
                System.out.print(" ");
            }

            // decrementing k after each loop
            k = k - 2;

            //  inner loop to handle number of columns
            //  values changing acc. to outer loop
            for(j=0; j<=i; j++)
            {
                // printing stars
                System.out.print("* ");
            }

            // ending line after each row
            System.out.println();
        }
    }

    // Driver Function
    public static void main(String args[])
    {
        int n = 5;
        printStars(n);
    }
}
```

Output:

```
        *
      * *
    * * *
  * * * *
* * * * *
```

- **Printing Triangle**

```
import java.io.*;

// Java code to demonstrate star pattern
public class GeeksForGeeks
{
    // Function to demonstrate printing pattern
    public static void printTriagle(int n)
    {
        // number of spaces
        int k = 2*n - 2;

        // outer loop to handle number of rows
        //  n in this case
        for (int i=0; i<n; i++)
        {

            // inner loop to handle number spaces
            // values changing acc. to requirement
            for (int j=0; j<k; j++)
            {
                // printing spaces
                System.out.print(" ");
            }

            // decrementing k after each loop
            k = k - 1;

            //  inner loop to handle number of columns
            //  values changing acc. to outer loop
            for (int j=0; j<=i; j++ )
            {
                // printing stars
                System.out.print("* ");
            }

            // ending line after each row
            System.out.println();
        }
    }

    // Driver Function
    public static void main(String args[])
    {
        int n = 5;
        printTriagle(n);
    }
}
```

Output:

```
    *
   * *
  * * *
 * * * *
* * * * *
```

- **Number Pattern**

```
import java.io.*;

// Java code to demonstrate number pattern
public class GeeksForGeeks
{
    // Function to demonstrate printing pattern
    public static void printNums(int n)
    {
        int i, j,num;

        // outer loop to handle number of rows
        //  n in this case
        for(i=0; i<n; i++)
        {
            // initialising starting number
            num=1;

            //  inner loop to handle number of columns
            //  values changing acc. to outer loop
            for(j=0; j<=i; j++)
            {
                // printing num with a space
                System.out.print(num+ " ");

                //incrementing value of num
                num++;
            }

            // ending line after each row
            System.out.println();
        }
    }

    // Driver Function
    public static void main(String args[])
    {
        int n = 5;
        printNums(n);
    }
}
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

- **Numbers without re assigning**

```
import java.io.*;

// Java code to demonstrate star pattern
public class GeeksForGeeks
{
    // Function to demonstrate printing pattern
    public static void printNums(int n)
    {
        // initialising starting number
        int i, j, num=1;

        // outer loop to handle number of rows
        // n in this case
        for(i=0; i<n; i++)
        {
            // without re assigning num
            // num = 1;
            for(j=0; j<=i; j++)
            {
                // printing num with a space
                System.out.print(num+ " ");

                // incrementing num at each column
                num = num + 1;
            }

            // ending line after each row
            System.out.println();
        }
    }

    // Driver Function
    public static void main(String args[])
    {
        int n = 5;
        printNums(n);
    }
}
```

Output:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
```

This article is contributed by **Nikhil Meherwal(S. Shafaq)**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# Socket Programming in Java

This article describes a very basic one-way Client and Server setup where a Client connects, sends messages to server and the server shows them using socket connection. There's a lot of low-level stuff that needs to happen for these things to work but the Java API networking package (java.net) takes care of all of that, making network programming very easy for programmers.

**Client Side Programming**

**Establish a Socket Connection**

To connect to other machine we need a socket connection. A socket connection means the two machines have information about each other's network location (IP Address) and TCP port.The java.net.Socket class represents a Socket. To open a socket:

```
Socket socket = new Socket("127.0.0.1", 5000)
```

- First argument – **IP address of Server**. ( 127.0.0.1  is the IP address of localhost, where code will run on single stand-alone machine).
- Second argument – **TCP Port**. (Just a number representing which application to run on a server. For example, HTTP runs on port 80. Port number can be from 0 to 65535)

**Communication**

To communicate over a socket connection, streams are used to both input and output the data.

**Closing the connection**

The socket connection is closed explicitly once the message to server is sent.

*In the program, Client keeps reading input from user and sends to the server until "Over" is typed.*

**Java Implementation**

```
// A Java program for a Client
import java.net.*;
import java.io.*;

public class Client
{
    // initialize socket and input output streams
    private Socket socket            = null;
    private DataInputStream  input   = null;
    private DataOutputStream out     = null;

    // constructor to put ip address and port
    public Client(String address, int port)
    {
        // establish a connection
        try
        {
            socket = new Socket(address, port);
            System.out.println("Connected");

            // takes input from terminal
            input  = new DataInputStream(System.in);

            // sends output to the socket
            out    = new DataOutputStream(socket.getOutputStream());
        }
        catch(UnknownHostException u)
        {
            System.out.println(u);
        }
        catch(IOException i)
        {
            System.out.println(i);
        }

        // string to read message from input
        String line = "";

        // keep reading until "Over" is input
        while (!line.equals("Over"))
        {
            try
            {
                line = input.readLine();
                out.writeUTF(line);
            }
            catch(IOException i)
            {
                System.out.println(i);
            }
        }

        // close the connection
        try
        {
            input.close();
            out.close();
            socket.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Client client = new Client("127.0.0.1", 5000);
    }
}
```

**Server Programming**

**Establish a Socket Connection**

To write a server application two sockets are needed.

- A ServerSocket which waits for the client requests (when a client makes a new Socket())
- A plain old Socket socket to use for communication with the client.

**Communication**

getOutputStream() method is used to send the output through the socket.

**Close the Connection**

After finishing,  it is important to close the connection by closing the socket as well as input/output streams.

```
// A Java program for a Server
import java.net.*;
import java.io.*;

public class Server
{
    //initialize socket and input stream
    private Socket          socket   = null;
    private ServerSocket    server   = null;
    private DataInputStream in       =  null;

    // constructor with port
    public Server(int port)
    {
        // starts server and waits for a connection
        try
        {
            server = new ServerSocket(port);
            System.out.println("Server started");

            System.out.println("Waiting for a client ...");

            socket = server.accept();
            System.out.println("Client accepted");

            // takes input from the client socket
            in = new DataInputStream(
                new BufferedInputStream(socket.getInputStream()));

            String line = "";

            // reads message from client until "Over" is sent
            while (!line.equals("Over"))
            {
                try
                {
                    line = in.readUTF();
                    System.out.println(line);

                }
                catch(IOException i)
                {
                    System.out.println(i);
                }
            }
            System.out.println("Closing connection");

            // close connection
            socket.close();
            in.close();
        }
        catch(IOException i)
        {
            System.out.println(i);
        }
    }

    public static void main(String args[])
    {
        Server server = new Server(5000);
    }
}
```

**Important Points**

- Server application makes a ServerSocket on a specific port which is 5000. This starts our Server listening for client requests coming in for port 5000.
- Then Server makes a new Socket to communicate with the client.

```
socket = server.accept()
```

- The accept() method blocks(just sits there) until a client connects to the server.
- Then we take input from the socket using getInputStream() method. Our Server keeps receiving messages until the Client sends “Over”.
- After we’re done we close the connection by closing the socket and the input stream.
- To run the Client and Server application on your machine, compile both of them. Then first run the server application and then run the Client application.

**To run on Terminal or Command Prompt**

Open two windows one for Server and another for Client

1. First run the Server application as ,

```
$ java Server
```

Server started
Waiting for a client …

2. Then run the Client application on another terminal as,

```
$ java Client
```

It will show – Connected and the server accepts the client and shows,

Client accepted

3. Then you can start typing messages in the Client window. Here is a sample input to the Client

```
Hello
I made my first socket connection
Over
```

Which the Server simultaneously receives and shows,

```
Hello
I made my first socket connection
Over
Closing connection
```

Notice that sending “Over” closes the connection between the Client and the Server just like said before.

**If you’re using Eclipse or likes of such-**

1. Compile both of them on two different terminals or tabs
2. Run the Server program first
3. Then run the Client program
4. Type messages in the Client Window which will be received and showed by the Server Window simultaneously.
5. Type Over to end.

This article is contributed by **Souradeep Barua**. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

# URL class in Java with Examples

The URL class is the gateway to any of the resource available on internet. A Class URL represents a Uniform Resource Locator, which is a pointer to a “resource” on the World Wide Web. A resource can point to a simple file or directory, or it can refer to a more complicated object, such as a query to a database or to a search engine

**What is a URL?**
As many of you must be knowing that Uniform Resource Locator-URL is a string of text that identifies all the resources on Internet, telling us the address of the resource, how to communicate with it and retrieve something from it.

A Simple URL looks like:

**Components of a URL:-**
A URL can have many forms. The most general however follows three-components system-

1. **Protocol:** HTTP is the protocol here
2. **Hostname:** Name of the machine on which the resource lives.
3. **File Name:** The path name to the file on the machine.
4. **Port Number:** Port number to which to connect (typically optional).

**Some constructors for URL class:-**

1. **URL(String address) throws MalformedURLException:** It creates a URL object from the specified String.
2. **URL(String protocol, String host, String file):** Creates a URL object from the specified protcol, host, and file name.
3. **URL(String protocol, String host, int port, String file):** Creates a URL object from protocol, host, port and file name.
4. **URL(URL context, String spec):** Creates a URL object by parsing the given spec in the given context.
5. **URL(String protocol, String host, int port, String file, URLStreamHandler handler):-**
   Creates a URL object from the specified protocol, host, port number, file, and handler.
6. **URL(URL context, String spec, URLStreamHandler handler):-**
   Creates a URL by parsing the given spec with the specified handler within a specified context.

**Sample Program:**

```
// Java program to demonstrate working of URL
import java.net.MalformedURLException;
import java.net.URL;
```

```
public class URLclass1
{
    public static void main(String[] args)
                throws MalformedURLException
    {

        // creates a URL with string representation.
        URL url1 =
        new URL("https://www.google.co.in/?gfe_rd=cr&ei=ptYq" +
                "WK26I4fT8gfth6CACg#q=geeks+for+geeks+java");

        // creates a URL with a protocol,hostname,and path
        URL url2 = new URL("http", "www.geeksforgeeks.org",
                    "/jvm-works-jvm-architecture/");

        URL url3 = new URL("https://www.google.co.in/search?"+
                    "q=gnu&rlz=1C1CHZL_enIN71" +
                    "4IN715&oq=gnu&aqs=chrome..69i57j69" +
                    "i60l5.653j0j7&sourceid=chrome&ie=UTF" +
                    "-8#q=geeks+for+geeks+java");

        // print the String representation of the URL.
        System.out.println(url1.toString());
        System.out.println(url2.toString());
        System.out.println();
        System.out.println("Different components of the URL3-");

        // retrieve the protocol for the URL
        System.out.println("Protocol:- " + url3.getProtocol());

        // retrieve the hostname of the url
        System.out.println("Hostname:- " + url3.getHost());

        // retrieve the defalut port
        System.out.println("Default port:- " +
                                url3.getDefaultPort());

        // retrieve the query part of URL
        System.out.println("Query:- " + url3.getQuery());

        // retrive the path of URL
        System.out.println("Path:- " + url3.getPath());

        // retrive the file name
        System.out.println("File:- " + url3.getFile());

        // retrieve the reference
        System.out.println("Reference:- " + url3.getRef());
    }
}
```

Output:

```
https://www.google.co.in/?gfe_rd=cr&ei=ptYqWK26I4fT8gfth6CACg#q=geeks+for+geeks+java
http://www.geeksforgeeks.org/jvm-works-jvm-architecture/

Different components of the URL3-
Protocol:- https
Hostname:- www.google.co.in
Default port:- 443
Query:- q=gnu&rlz=1C1CHZL_enIN714IN715&oq=gnu&aqs=chrome..69i57j69i60l5.653j0j7&sourceid=chrome&ie=UTF-8
Path:- /search
File:- /search?q=gnu&rlz=1C1CHZL_enIN714IN715&oq=gnu&aqs=chrome..69i57j69i60l5.653j0j7&sourceid=chrome&ie=UTF-8
Reference:- q=geeks+for+geeks+java
```

Explanation of some methods used in above program are as follows:-

1. **public String toString():** As in any class, toString() returns the string representation of the given URL object.
2. **public String getAuthority():** returns the authority part of URL or null if empty.
3. **public String getPath():** returns the path of the URL, or null if empty.
4. **public String getQuery():** returns the query part of URL. Query is the the part after the '?' in the URL. Whenever logic is used to display the result, there would be a query field in URL. It is similar to querying a database.
5. **public String getHost():** return the hostname of the URL in IPv6 format.
6. **public String getFile():** returns the file name.
7. **public String getRef():** Returns the reference of the URL object. Usually the reference is the part marked by a '#' in the URL. You can see the working exaple by querying anything on google and seeing the part after '#'.
8. **public int getPort():** returns the port associated with the protocol specified by the URL.
9. **public int getDefaultPort:** returns the default port used.
10. **public String getProtocol():** returns the protocol used by the URL.

**References**
http://docs.oracle.com/javase/7/docs/api/java/net/URL.html#getAuthority()

## Reading from a URL using URLConnection Class

In the previous article, we saw how we can create a URL object and get the information of any resource on the internet. But barely getting the state-information is not the true motive of a real world application. To retrieve the information, process it and sending the results back to the server, or just display the required information retrieved from the server is what we are aiming at. Consider, for example a small application which asks for a movie name from user and in turn returns the "imdb" rating of the movie or return all the links related to that movie. All of this can be achieved using the URLConnection class.

**What is URLConnection class?**
URLConnection is an abstract class whose subclasses form the link between the user application and any resource on the web. We can use it to read/write from/to any resource referenced by a URL object.

There are mainly two subclass that extends the URLConnection class-

- **HttpURLConnection:** If we are connecting to any url which uses "http" as its protocol, then HttpURLConnection class is used.

- **JarURLConnection:** If however, we are trying to establish a connection to a jar file on the web, then JarURLConnection is used.

Once the connection is established and we have a URLConnection object, we can use it to read or write or get further information about when was the page/file last modified, content length etc.

**Important Methods**

- **URLConnection openConnection():** opens the connection to the specified URL.
- **Object getContent():** retrieves the content of the URLConnection.
- **Map<String,List> getHeaderFields():** returns the map containing the values of various header fields in the http header.
- **getContentEncoding():** Returns the value of the content-encoding header field.
- **getContentLength():** returns the length of the content header field.
- **getDate():** returns value of date in header field.
- **getHeaderField(int i):** returns the value of $i^{th}$ index of header.
- **getHeaderField(String field):** returns the value of field named "field" in header
  To get the list of all header fields, read this.
- **OutputStream getOutputStream():** returns the output stream to this connection.
- **InputStream getInputStream():** returns the input stream to this open connection.
- **setAllowUserInteraction(boolean):** Setting this true means a user can interact with the page. Default value is true.
- **setDefaultUseCaches(boolean):** Sets the default value of useCache field as the given value.
- **setDoInput(boolean):** sets if the user is allowed to take input or not.
- **setDoOutput(boolean):** sets if the user is allowed to write on the page. Default value is false since most of the url dont allow to write.

Lets look at a sample program, which uses the above methods to display the header fields and also print the source code of entire page on to the console window.

```
//Java Program  to illustrate reading and writing
// in URLConnection Class
import java.io.*;
import java.net.*;
import java.util.ArrayList;
import java.util.Date;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class URLConnectionclass
{
 public static void main(String[] args)
 {
  try
  {
   URL url = new URL("http://www.geeksforgeeks.org/java");

   //open the connection to the above URL.
   URLConnection urlcon = url.openConnection();

   //Executing the below line would print the value of
   // Allow User Interaction field.
   // System.out.println(urlcon.getAllowUserInteraction());

   //Executing the below line would print
   // the value of Content Type field.
   // System.out.println(urlcon.getContentType());

   //Executing the below line would print the value
   // of URL of the given connection.
   // System.out.println(urlcon.getURL());

   //Executing the below line would
   // print the value of Do Input field.
   // System.out.println(urlcon.getDoInput());

   //Executing the below line would
   // print the value of Do Output field.
   // System.out.println(urlcon.getDoOutput());

   //Executing the below line would
   // print the value of Last Modified field.
   // System.out.println(new Date(urlcon.getLastModified()));

   //Executing the below line would
   // print the value of Content Encoding field.
   // System.out.println(urlcon.getContentEncoding());

   //To get a map of all the fields of http header
   Map<String, List<String>> header = urlcon.getHeaderFields();

   //print all the fields along with their value.
   for (Map.Entry<String, List<String>> mp : header.entrySet())
   {
    System.out.print(mp.getKey() + " : ");
    System.out.println(mp.getValue().toString());
   }
   System.out.println();
   System.out.println("Complete source code of the URL is-");
   System.out.println("---------------------------------");

   //get the inputstream of the open connection.
   BufferedReader br = new BufferedReader(new InputStreamReader
          (urlcon.getInputStream()));
   String i;

   //print the source code line by line.
   while ((i = br.readLine()) != null)
   {
    System.out.println(i);
   }
  }

  catch (Exception e)
  {
   System.out.println(e);
  }
 }
}
```

Output:

```
Keep-Alive   :  [timeout=5, max=100]
null   :  [HTTP/1.1 200 OK]
Server   :  [Apache/2.4.18 (Ubuntu)]
Connection   :  [Keep-Alive]
Last-Modified   :  [Wed, 16 Nov 2016 06:49:55 GMT]
Date   :  [Wed, 16 Nov 2016 10:58:34 GMT]
Accept-Ranges   :  [bytes]
Cache-Control   :  [max-age=3]
ETag   :  ["10866-541657b07e4d7"]
Vary   :  [Accept-Encoding]
Expires   :  [Wed, 16 Nov 2016 10:58:37 GMT]
Content-Length   :  [67686]
Content-Type   :

Complete source code of the URL is-
---------------------------------------------------

...source code of the page...
```

**Steps involved in the above process:**

1. **URL Creation:** Create a URL object using any of the constructors given int
2. this article
3. **Create Object:** Invoke the openConnection() call to create the object of URLConnection.
4. **Display the Content:** Either use the above created object to display the information about the resource or to read/write contents of the file to console using bufferedReader and InputStream of the open connection using getInputStream() method.
5. **Close Stream:** Close the InputStream when done.

Reference-https://docs.oracle.com/javase/7/docs/api/java/net/URLConnection.html

This article is contributed by **Rishabh Mahrsee**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java

# A Group chat application in Java

In this post, a group chat application using MulticastSocket (Java Platform SE 7) class is discussed. A MulticastSocket is a (UDP) DatagramSocket, with additional capabilities for joining “groups” of other multicast hosts on the internet.

**Implementation**

```
import java.net.*;
import java.io.*;
import java.util.*;
public class GroupChat
{
    private static final String TERMINATE = "Exit";
    static String name;
    static volatile boolean finished = false;
    public static void main(String[] args)
    {
        if (args.length != 2)
            System.out.println("Two arguments required:
            <multicast-host> <port-number>");
        else
        {
            try
            {
                InetAddress group = InetAddress.getByName(args[0]);
                int port = Integer.parseInt(args[1]);
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter your name: ");
                name = sc.nextLine();
                MulticastSocket socket = new MulticastSocket(port);

                // Since we are deploying
                socket.setTimeToLive(0);
                this on localhost only (For a subnet set it as 1)
                socket.joinGroup(group);
                Thread t = new Thread(new
                ReadThread(socket,group,port));

                // Spawn a thread for reading messages
                t.start();

                // sent to the current group
                System.out.println("Start typing messages...\n");
                while(true)
                {
                    String message;
                    message = sc.nextLine();
                    if(message.equalsIgnoreCase(GroupChat.TERMINATE))
                    {
                        finished = true;
                        socket.leaveGroup(group);
                        socket.close();
                        break;
                    }
                    message = name + ": " + message;
                    byte[] buffer = message.getBytes();
                    DatagramPacket datagram = new
                    DatagramPacket(buffer,buffer.length,group,port);
                    socket.send(datagram);
                }
            }
            catch(SocketException se)
            {
                System.out.println("Error creating socket");
                se.printStackTrace();
            }
            catch(IOException ie)
            {
                System.out.println("Error reading/writing from/to
                socket");
                ie.printStackTrace();
            }
        }
    }
}
class ReadThread implements Runnable
{
    private MulticastSocket socket;
    private InetAddress group;
    private int port;
    private static final int MAX_LEN = 1000;
    ReadThread(MulticastSocket socket,InetAddress group,int port)
    {
        this.socket = socket;
        this.group = group;
        this.port = port;
    }

    @Override
    public void run()
    {
        while(!GroupChat.finished)
        {
                byte[] buffer = new byte[ReadThread.MAX_LEN];
                DatagramPacket datagram = new
                DatagramPacket(buffer,buffer.length,group,port);
                String message;
            try
            {
                socket.receive(datagram);
                message = new
                String(buffer,0,datagram.getLength(),"UTF-8");
                if(!message.startsWith(GroupChat.name))
                    System.out.println(message);
            }
            catch(IOException e)
            {
                System.out.println("Socket closed!");
            }
        }
    }
}
```

Save the file as GroupChat.java and compile it using javac and then run the program using two command line arguments as specified. A multicast host is specified by a class D IP address and by a standard UDP port number. Class D IP addresses are in the range 224.0.0.0 to 239.255.255.255, inclusive. The address 224.0.0.0 is reserved and should not be used.
Here is a sample output of the above program:

multicast socket api in java

multicast socket api in java1

multicast socket api in java12

We have used the multicast host IP address as 239.0.0.0 and the port number as 1234 (since the port numbers 0 through 1023 are reserved). There are 3 members in the group: Ironman, CaptainAmerica, and Groot. Start all three terminals first before sending the message, otherwise messages which are sent before starting the terminal are lost (since there is no facility of buffer incorporated to store the messages.) We need two threads in this application. One for accepting the user input (using the java.util.Scanner class) and the other for reading the messages sent from other clients. Hence I have separated the thread which does the reading work into ReadThreadclass. For leaving the group, any of the user can type in Exit to terminate the session.

The above program is executed on a single machine. Socket programming is meant for distributed programming. The same piece of code snippet when present on different machines which have Java installed can satisfy that requirement. This is just the bare bones service logic. The project would be even more fascinating if the front-end is developed. You can use Java's AWT (Abstract Window Toolkit) or its advanced counterpart, Java Swing to develop the front end. Since this wouldn't be part of Socket programming I'm leaving it untouched without getting into the details.

**Additional points:**

- You can incorporate network security feature by performing encryption before sending the message over the network.
- Primitive techniques such as Caesar cipher or advanced methods such as RSA can be used to perform encryption-decryption. You can try using Java's RMI (Remote Method Invocation) to perform the same task.
- Here, you can leverage the abstraction offered by Java to maximum extent. However, if your primary objective is efficiency, then Socket programming is the best choice. Since it doesn't require any run time

support, it is a bit faster compared to RMI.

This article is contributed by **Aditya ch**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

GBlog
Java
Project

# Networking in Java | Set 1 (InetAddress class)

Java provides classes which programming on internet or TCP/IP network. Data transfer between Java programs, running on different systems, is done using IO streams. All classes are made available through java.net package.

**InetAddress class**

This class represents an IP address and domain name. InetAddress has factory methods, i.e. static methods that return instance of class, which return an object of InetAddress for the given name with the required details. This class provides methods to get name and host address of a system. It provides the following methods:

| **Method** | **Meaning** |
|---|---|
| static InetAddress[] getAllByName(String host) | Determines all the IP addresses of the given host's name. |
| static InetAddress[] getByName(String host) | Determines the IP address of the given host's name. |
| String getHostAddress() | Returns the IP address string "%d.%d.%d.%d". |
| String getHostName() | Returns the hostname for this address. |
| Static InetAddress getLocalHost() | Returns the local host. |

The following program uses InetAddress class to get IP address of the given domain name. When the program is run on a system connected to the Internet, it gives the IP address(es) of the domain given.

```
// A Java program to demonstrate working of InetAddress class
// This program finds IP address for a domain name.
import java.net.*;

public class GetIPAddress
{
    public static void main(String args[]) throws Exception
    {
        String url = "www.google.com";
        try
        {
            // Get IP addresses related to the domain
            InetAddress ips[] = InetAddress.getAllByName(url);

            // Display ip addresses
            System.out.println("IP Address(es)");
            for (InetAddress addr:ips)
                System.out.println(addr.getHostAddress());
        }
        catch(Exception ex)
        {
            System.out.println("host not found");
        }
    }
}
```

**Input:**

**Output:**

```
IP Address(es)
172.217.4.68
2607:f8b0:4006:809:0:0:0:2004
```

This article is contributed by **Aparna Vadlamani.** Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.
Category: Computer Networks Java

# Output of Java Program | Set 1

**Difficulty Level:** Rookie

Predict the output of following Java Programs.

**Program 1**

```
// filename Main.java
class Test {
    protected int x, y;
}

class Main {
    public static void main(String args[]) {
        Test t = new Test();
        System.out.println(t.x + " " + t.y);
    }
}
```

Output

0 0

In Java, a protected member is accessible in all classes of same package and in inherited classes of other packages. Since Test and Main are in same package, no access related problem in the above program. Also, the default constructors initialize integral variables as 0 in Java (See this GFact for more details). That is why we get output as 0 0.

**Program 2**

```
// filename Test.java
```

```
class Test {
    public static void main(String[] args) {
        for(int i = 0; 1; i++) {
            System.out.println("Hello");
            break;
        }
    }
}
```

Output: Compiler Error

There is an error in condition check expression of for loop. Java differs from C++(or C) here. C++ considers all non-zero values as true and 0 as false. Unlike C++, an integer value expression cannot be placed where a boolean is expected in Java. Following is the corrected program.

```
// filename Test.java
class Test {
    public static void main(String[] args) {
        for(int i = 0; true; i++) {
            System.out.println("Hello");
            break;
        }
    }
}
// Output: Hello
```

**Program 3**

```
// filename Main.java
class Main {
    public static void main(String args[]) {
        System.out.println(fun());
    }
    int fun() {
        return 20;
    }
}
```

Output: Compiler Error

Like C++, in Java, non-static methods cannot be called in a static method. If we make fun() static, then the program compiles fine without any compiler error. Following is the corrected program.

```
// filename Main.java
class Main {
    public static void main(String args[]) {
        System.out.println(fun());
    }
    static int fun() {
        return 20;
    }
}
// Output: 20
```

**Program 4**

```
// filename Test.java
class Test {
   public static void main(String args[]) {
      System.out.println(fun());
   }
   static int fun() {
      static int x= 0;
      return ++x;
   }
}
```

Output: Compiler Error

Unlike C++, static local variables are not allowed in Java. See this GFact for details. We can have class static members to count number of function calls and other purposes that C++ local static variables serve. Following is the corrected program.

```
class Test {
   private static int x;
   public static void main(String args[]) {
      System.out.println(fun());
   }
   static int fun() {
      return ++x;
   }
}
// Output: 1
```

Please write comments if you find any of the answers/explanations incorrect, or want to share more information about the topics discussed above.

## GATE CS Corner    Company Wise Coding Practice

Output
Java
Java-Output

# Output of Java Program | Set 2

Predict the output of following Java programs.

**Question 1**

```
package main;

class Base {
    public void Print() {
        System.out.println("Base");
    }
}

class Derived extends Base {
    public void Print() {
        System.out.println("Derived");
    }
}
```

```
class Main{
    public static void DoPrint( Base o ) {
        o.Print();
    }
    public static void main(String[] args) {
        Base x = new Base();
        Base y = new Derived();
        Derived z = new Derived();
        DoPrint(x);
        DoPrint(y);
        DoPrint(z);
    }
}
```

Output:

```
Base
Derived
Derived
```

Predicting the first line of output is easy. We create an object of type Base and call DoPrint(). DoPrint calls the print function and we get the first line.

DoPrint(y) causes second line of output. Like C++, assigning a derived class reference to a base class reference is allowed in Java. Therefore, the expression Base y = new Derived() is a valid statement in Java. In DoPrint(), o starts referring to the same object as referred by y. Also, unlike C++, functions are virtual by default in Java. So, when we call o.print(), the print() method of Derived class is called due to run time polymorphism present by default in Java.

DoPrint(z) causes third line of output, we pass a reference of Derived type and the print() method of Derived class is called again. The point to note here is: unlike C++, object slicing doesn't happen in Java. Because non-primitive types are always assigned by reference.

**Question 2**

```
package main;

// filename Main.java
class Point {
    protected int x, y;

    public Point(int _x, int _y) {
        x = _x;
        y = _y;
    }
}

public class Main {
    public static void main(String args[]) {
      Point p = new Point();
      System.out.println("x = " + p.x + ", y = " + p.y);
    }
}
```

Output:
Compiler Error
In the above program, there are no access permission issues because the Test and Main are in same package and protected members of a class can be accessed in other classes of same package. The problem with the code is: there is not default constructor in Point. Like C++, if we write our own parametrized constructor then Java compiler doesn't create the default constructor. So there are following two changes to Point class that can fix the above program.
1) Remove the parametrized constructor.
2) Add a constructor without any parameter.
Java doesn't support default arguments, so that is not an option.

Please write comments if you find any of the answers/explanations incorrect, or want to share more information about the topics discussed above.

## GATE CS Corner    Company Wise Coding Practice

Output
Java
Java-Output

# Output of Java Program | Set 3

Predict the output of following Java Programs:

```
// filename: Test.java
class Test {
    int x  = 10;
    public static void main(String[] args) {
         Test t = new Test();
         System.out.println(t.x);
    }
}
```

The program works fine and prints 10. Unlike C++, in Java, members can initialized with declaration of class. This initialization works well when the initialization value is available and the initialization can be put on one line (See this for more details). For example, the following program also works fine.

```
// filename: Test.java
class Test {
    int y = 2;
    int x  = y+2;
    public static void main(String[] args) {
         Test m = new Test();
         System.out.println("x = " + m.x + ", y = " + m.y);
    }
}
```

Output of the above program is "x = 4, y = 2". y is initialized first, then x is initialized as y + 2. So the value of x becomes 4.

What happen when a member is initialized in class declaration and constructor both? Consider the following program.

```
// filename: Test.java
public class Test
{
    int x = 2;
    Test(int i) { x = i; }
    public static void main(String[] args) {
        Test t = new Test(5);
        System.out.println("x = " + t.x);
    }
}
```

Output of the above program is “x = 5”. The initialization with class declaration in Java is like initialization using Initializer List in C++. So, in the above program, the value assigned inside the constructor overwrites the previous value of x which is 2, and x becomes 5.

As an exercise, predict the output of following program.

```
// filename: Test2.java
class Test1 {
    Test1(int x) {
        System.out.println("Constructor called " + x);
    }
}

// This class contains an instance of Test1
class Test2 {
    Test1 t1 = new Test1(10);

    Test2(int i) { t1 = new Test1(i); }

    public static void main(String[] args) {
        Test2 t2 = new Test2(5);
    }
}
```

Please write comments if you find any of the answers/explanations incorrect, or want to share more information about the topics discussed above.

## GATE CS Corner    Company Wise Coding Practice

Output
Java
Java-Output

# Output of Java Program | Set 4

Predict the output of following Java Programs:

**Question 1**

```
// file name: Main.java

class Base {
    protected void foo() {}
}
class Derived extends Base {
    void foo() {}
}
public class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.foo();
    }
}
```

Output: Compiler Error
foo() is protected in Base and default in Derived. Default access is more restrictive. When a derived class overrides a base class function, more restrictive access can’t be given to the overridden function. If we make foo() public, then the program works fine without any error. The behavior in C++ is different. C++ allows to give more restrictive access to derived class methods.

**Question 2**

```
// file name: Main.java

class Complex {
    private double re, im;
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
    Complex(Complex c) {
        re = c.re;
        im = c.im;
    }
}

public class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex();
        Complex c2 = new Complex(c1);
        System.out.println(c2);
    }
}
```

Output: Compiler Error in line “Complex c1 = new Complex();”
In Java, if we write our own copy constructor or parameterized constructor, then compiler doesn’t create the default constructor. This behavior is same as C++.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Output
Java
Java-Output
Output of Java Program

# Output of Java program | Set 5

Predict the output of following Java Programs.

**Program 1:**

```
// Main.java
public class Main
{
    public static void gfg(String s)
    {
        System.out.println("String");
    }
    public static void gfg(Object o)
    {
        System.out.println("Object");
    }

    public static void main(String args[])
    {
        gfg(null);
    }
} //end class
```

**Output**:

```
String
```

**Explanation** : In case of method overloading, the most specific method is chosen at compile time. As ‘java.lang.String’ is a more specific type than ‘java.lang.Object’. In this case the method which takes ‘String’ as a parameter is choosen.

**Program 2:**

```
// Main.java
public class Main
{
    public static void gfg(String s)
    {
        System.out.println("String");
    }
    public static void gfg(Object o)
    {
        System.out.println("Object");
    }
    public static void gfg(Integer i)
    {
        System.out.println("Integer");
    }

    public static void main(String args[])
    {
        gfg(null);
    }
} //end class
```

**Output:**

```
Compile Error at line 19.
```

**Explanation:** In this case of method Overloading, the most specific method is choosen at compile time.
As ‘java.lang.String’ and ‘java.lang.Integer’ is a more specific type than ‘java.lang.Object’,but between ‘java.lang.String’ and ‘java.lang.Integer’ none is more specific.
In this case the Java is unable to decide which method to call.

**Program 3:**

```
// Main.java
public class Main
{
    public static void main(String args[])
    {
        String s1 = "abc";
        String s2 = s1;
        s1 += "d";
        System.out.println(s1 + " " + s2 + " " + (s1 == s2));

        StringBuffer sb1 = new StringBuffer("abc");
        StringBuffer sb2 = sb1;
        sb1.append("d");
        System.out.println(sb1 + " " + sb2 + " " + (sb1 == sb2));
    }
} //end class
```

Output:

```
abcd abc false
abcd abcd true
```

**Explanation :** In Java, String is immutable and string buffer is mutable.
So string s2 and s1 both pointing to the same string abc. And, after making the changes the string s1 points to abcd and s2 points to abc, hence false. While in string buffer, both sb1 and sb2 both point to the same object. Since string buffer are mutable, making changes in one string also make changes to the other string. So both string still pointing to the same object after making the changes to the object (here sb2).

**Program 4:**

```
// Main.java
public class Main
{
    public static void main(String args[])
    {
        short s = 0;
        int x = 07;
        int y = 08;
        int z = 112345;

        s += z;
        System.out.println("" + x + y + s);
    }
} //end class
```

**Output:**

```
Compile Error at line 8
```

**Explanation:**
1. In Line 12 The “” in the println causes the numbers to be automatically cast as strings. So it doesn’t do addition, but appends together as string.
2. In Line11 the += does an automatic cast to a short. However the number 123456 can’t be contained within a short, so you end up with a negative value (-7616).
3. Those other two are red herrings however as the code will never compile due to line 8. Any number beginning with zero is treated as an octal number (which is 0-7).

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

# Output of Java Program | Set 6

**Difficulty level :** Intermediate

Predict the output of following Java Programs.

**Program 1:**

```
class First
{
    public First() {  System.out.println("a"); }
}

class Second extends First
{
    public Second()  {  System.out.println("b"); }
}

class Third extends Second
{
    public Third()   {  System.out.println("c"); }
}

public class MainClass
{
    public static void main(String[] args)
    {
        Third c = new Third();
    }
}
```

Output:

```
a
b
c
```

**Explanation:**
While creating a new object of 'Third' type, before calling the default constructor of Third class, the default constructor of super class is called i.e, Second class and then again before the default constructor of super class, default constructor of First class is called. And hence gives such output.

**Program 2:**

```
class First
{
    int i = 10;

    public First(int j)
    {
        System.out.println(i);
        this.i = j * 10;
    }
}

class Second extends First
{
    public Second(int j)
    {
        super(j);
        System.out.println(i);
        this.i = j * 20;
    }
}

public class MainClass
{
    public static void main(String[] args)
    {
        Second n = new Second(20);
        System.out.println(n.i);
    }
}
```

Output:

```
10
200
400
```

**Explanation:**
Since in 'Second' class it doesn't have its own 'i', the variable is inherited from the super class. Also, the constructor of parent is called when we create an object of Second.

**Program 3:**

```
import java.util.*;
class I
{
    public static void main (String[] args)
    {
        Object i = new ArrayList().iterator();
        System.out.print((i instanceof List) + ", ");
        System.out.print((i instanceof Iterator) + ", ");
        System.out.print(i instanceof ListIterator);
    }
}
```

Output:

```
false, true, false
```

**Explanation:**
The iterator() method returns an iterator over the elements in the list in proper sequence, it doesn't return a List or a ListIterator object. A ListIterator can be obtained by invoking the listIterator method.

**Program 4:**

```
class ThreadEx extends Thread
{
    public void run()
    {
        System.out.print("Hello...");
    }
    public static void main(String args[])
    {
        ThreadEx T1 = new ThreadEx();
        T1.start();
        T1.stop();
        T1.start();
    }
}
```

Output:

```
Run Time Exception
```

**Explanation:**
Exception in thread "main" java.lang.IllegalThreadStateException at java.lang.Thread.start
Thread cannot be started twice.

This article is contributed by **Pratik Agrawal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Output
Java-Output

# Output of Java Program | Set 7

**Difficulty level :** Intermediate

Predict the output of following Java Programs.

**Program 1 :**

```
public class Calculator
{
    int num = 100;
    public void calc(int num)  { this.num = num * 10;  }
    public void printNum()     { System.out.println(num); }

    public static void main(String[] args)
    {
        Calculator obj = new Calculator();
        obj.calc(2);
        obj.printNum();
    }
}
```

**Options :**
A) 20
B) 100
C) 1000
D) 2
**Answer : A) 20**
**Explanation :** Here the class instance variable name(num) is same as *calc()* method local variable name(num). So for referencing class instance variable from *calc()* method, **this** keyword is used. So in statement **this.num = num * 10**, *num* represents local variable of the method whose value is 2 and *this.num* represents class instance variable whose initial value is 100. Now in *printNum()* method, as it has no local variable whose name is same as class instance variable, so we can directly use *num* to reference instance variable, although *this.num* can be used.

**Program 2 :**

```
public class MyStuff
{
    String name;

    MyStuff(String n) {  name = n;  }

    public static void main(String[] args)
    {
        MyStuff m1 = new MyStuff("guitar");
        MyStuff m2 = new MyStuff("tv");
        System.out.println(m2.equals(m1));
    }

    @Override
    public boolean equals(Object obj)
    {
        MyStuff m = (MyStuff) obj;
        if (m.name != null)  { return true;  }
        return false;
    }
}
```

**Options :**
A) The output is true and MyStuff fulfills the Object.equals() contract.
B) The output is false and MyStuff fulfills the Object.equals() contract.
C) The output is true and MyStuff does NOT fulfill the Object.equals() contract.
D) The output is false and MyStuff does NOT fulfill the Object.equals() contract.

**Answer :** C) The output is true and MyStuff does NOT fulfill the Object.equals() contract.

**Explanation :** As *equals(Object obj)* method in Object class, compares two objects on the basis of equivalence relation. But here we are just confirming that the object is null or not, So it doesn't fulfill Object.equals() contract. As *m1* is not null, true will be printed.

**Program 3 :**

```
class Alpha
{
    public String type = "a ";
    public Alpha() {  System.out.print("alpha "); }
}

public class Beta extends Alpha
{
    public Beta()  {  System.out.print("beta ");  }

    void go()
    {
        type = "b ";
        System.out.print(this.type + super.type);
    }

    public static void main(String[] args)
    {
        new Beta().go();
    }
}
```

**Options :**

A) alpha beta b b

B) alpha beta a b

C) beta alpha b b

D) beta alpha a b

**Answer :** A) alpha beta b b

**Explanation :** The statement **new Beta().go()** executes in two phases. In first phase *Beta* class constructor is called. There is no instance member present in *Beta* class. So now *Beta* class constructor is executed. As *Beta* class extends *Alpha* class, so call goes to *Alpha* class constructor as first statement by default(Put by the compiler) is **super()** in the *Beta* class constructor. Now as one instance variable(*type*) is present in *Alpha* class, so it will get memory and now *Alpha* class constructor is executed, then call return to *Beta* class constructor next statement. So *alpha beta* is printed.
In second phase *go()* method is called on this object. As there is only one variable(*type*) in the object whose value is *a*. So it will be changed to *b* and printed two times. The super keyword keyword here is of no use.

**Program 4 :**

```
public class Test
{
    public static void main(String[] args)
    {
        StringBuilder s1 = new StringBuilder("Java");
        String s2 = "Love";
        s1.append(s2);
        s1.substring(4);
        int foundAt = s1.indexOf(s2);
        System.out.println(foundAt);
    }
}
```

**Options :**

A) -1

B) 3

C) 4

D) A **StringIndexOutOfBoundsException** is thrown at runtime.

**Answer :** C) 4

**Explanation :** *append(String str)* method,concatenate the str to *s1*. The *substring(int index)* method return the String from the given index to the end. But as there is no any String variable to store the returned string,so it will be destroyed.Now *indexOf(String s2)* method return the index of first occurrence of *s2*. So 4 is printed as s1="JavaLove".

**Program 5 :**

```
class Writer
{
    public  static void write()
    {
        System.out.println("Writing...");
    }
}
class Author extends Writer
{
    public  static void write()
    {
        System.out.println("Writing book");
    }
}

public class Programmer extends Author
{
    public  static void write()
    {
        System.out.println("Writing code");
    }

    public static void main(String[] args)
    {
        Author a = new Programmer();
        a.write();
    }
}
```

**Options :**

A) Writing…

B) Writing book

C) Writing code

D) Compilation fails

**Answer :** B) Writing book

**Explanation :** Since static methods can't be overridden, it doesn't matter which class object is created. As *a* is a *Author* referenced type, so always *Author* class method is called. If we remove *write()* method from *Author* class then *Writer* class method is called, as *Author* class extends *Writer* class.

This article is contributed by **Gaurav Miglani**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Java
Java-Output

# Output of Java Program | Set 8

**Difficulty level** : Intermediate

Predict the output of following Java Programs.

**Program 1:**

```
class GfG
{
    public static void main(String args[])
    {
        String s1 = new String("geeksforgeeks");
        String s2 = new String("geeksforgeeks");
        if (s1 == s2)
            System.out.println("Equal");
        else
            System.out.println("Not equal");
    }
}
```

Output:

```
Not equal
```

**Explanation:** Since, s1 and s2 are two different objects the references are not the same, and the == operator compares object reference. So it prints "Not equal", to compare the actual characters in the string .equals() method must be used.

**Program 2:**

```
class Person
{
    private void who()
    {
        System.out.println("Inside private method Person(who)");
    }

    public static void whoAmI()
    {
        System.out.println("Inside static method, Person(whoAmI)");
    }

    public void whoAreYou()
    {
        who();
        System.out.println("Inside virtual method, Person(whoAreYou)");
    }
}

class Kid extends Person
{
    private void who()
    {
        System.out.println("Kid(who)");
    }

    public static void whoAmI()
    {
        System.out.println("Kid(whoAmI)");
    }

    public void whoAreYou()
    {
        who();
        System.out.println("Kid(whoAreYou)");
    }
}
public class Gfg
{
    public static void main(String args[])
    {
        Person p = new Kid();
        p.whoAmI();
        p.whoAreYou();
    }
}
```

Output:

```
Inside static method, People(whoAmI)
Kid(who)
Kid(whoAreYou)
```

**Explanation:** Static binding (or compile time) happens for static methods. Here *p.whoAmI()* calls the static method so it is called during compile time hence results in static binding and prints the method in *People* class.
Whereas *p.whoAreYou()* calls the method in *Kid* class since by default Java takes it as a virual method i.e, dynamic binding.

**Program 3:**

```
class GfG
{
public static void main(String args[])
  {
    try
    {
        System.out.println("First statement of try block");
      int num=45/3;
      System.out.println(num);
    }
    catch(Exception e)
    {
      System.out.println("Gfg caught Exception");
    }
    finally
    {
      System.out.println("finally block");
    }
    System.out.println("Main method");
  }
}
```

Output:

```
First statement of try block
15
finally block
Main method
```

**Explanation:**
Since there is no exception, the catch block is not called, but the *finally* block is always executed after a try block whether the exception is handled or not.

**Program 4:**

```
class One implements Runnable
{
    public void run()
    {
        System.out.print(Thread.currentThread().getName());
    }
}
class Two implements Runnable
{
    public void run()
    {
        new One().run();
        new Thread(new One(),"gfg2").run();
        new Thread(new One(),"gfg3").start();
    }
}
class Three
{
    public static void main (String[] args)
    {
        new Thread(new Two(),"gfg1").start();
    }
}
```

Output:

```
gfg1gfg1gfg3
```

**Explanation :** Initially new Thread is started with name *gfg1* then in class Two the first run method runs the thread with the name *gfg1*, then after that a new thread is created calling run method but since a new thread can be created by calling start method only so the previous thread does the action and again *gfg1* is printed.Now a new thread is created by calling the start method so a new thread starts with *gfg3* name and hence prints *gfg3*.

This article is contributed by **Pratik Agarwal**. If you like GeeksforGeeks and would like to contribute, you can also write an article using contribute.geeksforgeeks.org or mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Corner    Company Wise Coding Practice

Java
Output
Java-Output

# Commonly Asked Java Programming Interview Questions | Set 1

**Why is Java called the 'Platform Independent Programming Language'?**

Platform independence means that execution of your program does not dependent on type of operating system(it could be any : Linux, windows, Mac ..etc). So compile code only once and run it on any System (In C/C++, we need to compile the code for every machine on which we run it). Java is both compiler(javac) and interpreter(jvm) based lauguage. Your java source code is first compiled into byte code using javac compiler. This byte code can be easily converted to equivalent machine code using JVM. JVM(Java Virtual Machine) is available in all operating systems we install. Hence, byte code generated by javac is universal and can be converted to machine code on any operating system, this is the reason why java is platform independent.

**Explain Final keyword in java?**

Final keyword in java is used to restrict usage of variable, class and method.

Variable: Value of Final variable is constant, you can not change it.
Method: you can't override a Final method.
Class: you can't inherit from Final class.

Refer this for details

**When is the super keyword used?**

super keyword is used to refer:

- immediate parent class constructor,
- immediate parent class variable,
- immediate parent class method.

Refer this for details.

**What is the difference between StringBuffer and String?**

String is an Immutable class, i.e. you can not modify its content once created. While StringBuffer is a mutable class, means you can change its content later. Whenever we alter content of String object, it creates a new string and refer to that,it does not modify the existing one. This is the reason that the performance with StringBuffer is better than with String.
Refer this for details.

**Why multiple inheritance is not supported in java?**

Java supports multiple inheritance but not through classes, it supports only through its interfaces. The reason for not supporting multiple inheritance is to avoid the conflict and complexity arises due to it and keep Java a Simple Object Oriented Language. If we recall this in C++, there is a special case of multiple inheritance (diamond problem) where you have a multiple inheritance with two classes which have methods in conflicts. So, Java developers decided to avoid such conflicts and didn't allow multiple inheritance through classes at all.

**Can a top level class be private or protected?**

Top level classes in java can't be private or protected, but inner classes in java can. The reason for not making a top level class as private is very obvious, because nobody can see a private class and thus they can not use it. Declaring a class as protected also doesn't make any sense. The only difference between default visibility and protected visibility is that we can use it in any package by inheriting it. Since in java there is no such concept of package inheritance, defining a class as protected is no different from default.

**What is the difference between 'throw' and 'throws' in Java Exception Handling?**

Following are the differences between two:

- throw keyword is used to throw Exception from any method or static block whereas throws is used to indicate that which Exception can possibly be thrown by this method
- If any method throws checked Exception, then caller can either handle this exception(using try catch block )or can re throw it by declaring another 'throws' clause in method declaration.
- throw clause can be used in any part of code where you feel a specific exception needs to be thrown to the calling method

E.g.
**throw**
throw new Exception("You have some exception")
throw new IOException("Connection failed!!")
**throws**
throws IOException, NullPointerException, ArithmeticException

**What is finalize() method?**

Unlike c++ , we don't need to destroy objects explicitly in Java. 'Garbage Collector' does that automatically for us. Garbage Collector checks if no references to an object exist, that object is assumed to be no longer required, and the memory occupied by the object can be freed. Sometimes an object can hold non-java resources such as file handle or database connection, then you want to make sure these resources are also released before object is destroyed. To perform such operation Java provide protected void finalize() in object class. You can override this method in your class and do the required tasks. Right before an object is freed, the java run time calls the finalize() method on that object. Refer this for more details.

**Difference in Set and List interface?**

Set and List both are child interface of Collection interface. There are following two main differences between them

- List can hold duplicate values but Set doesn't allow this.
- In List interface data is present in the order you inserted but in the case of Set insertion order is not preserved.

**What will happen if you put System.exit(0) on try or catch block? Will finally block execute?**

By Calling System.exit(0) in try or catch block, we can skip the finally block. System.exit(int) method can throw a SecurityException. If Sysytem.exit(0) exits the JVM without throwing that exception then finally block will not execute. But, if System.exit(0) does throw security exception then finally block will be executed.

- Practice Quizzes of Java
- Java Articles

This article is compiled by **Dharmesh Singh**.

You may like to see following:

Commonly Asked C Programming Interview Questions | Set 1
Commonly Asked C Programming Interview Questions | Set 2
Amazon's most asked interview questions
Microsoft's most asked interview questions
Accenture's most asked Interview Questions
Commonly Asked OOP Interview Questions
Commonly Asked C++ Interview Questions
Commonly asked DBMS interview questions | Set 1
Commonly asked DBMS interview questions | Set 2
Commonly Asked Operating Systems Interview Questions | Set 1
Commonly Asked Data Structure Interview Questions.
Commonly Asked Algorithm Interview Questions
Commonly asked Computer Networks Interview Questions
Top 10 algorithms in Interview Questions

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

# Commonly Asked Java Programming Interview Questions | Set 2

Frequently Asked Java Interview Questions | Set 1

**Can we Overload or Override static methods in java ?**

- **Overriding :** Overriding is related to run-time polymorphism. A subclass (or derived class) provides a specific implementation of a method in superclass (or base class) at runtime.
- **Overloading:** Overloading is related to compile time (or static) polymorphism. This feature allows different methods to have same name, but different signatures, especially number of input parameters and type of input paramaters.
- **Can we overload static methods?** The answer is **'Yes'**. We can have two ore more static methods with same name, but differences in input parameters
- **Can we Override static methods in java?** We can declare static methods with same signature in subclass, but it is not considered overriding as there won't be any run-time polymorphism. Hence the answer is **'No'**. Static methods cannot be overridden because method overriding only occurs in the context of dynamic (i.e. runtime) lookup of methods. Static methods (by their name) are looked up statically (i.e. at compile-time).

Read more

**Why the main method is static in java?**
The method is static because otherwise there would be ambiguity: which constructor should be called? Especially if your class looks like this:

```
public class JavaClass
{
  protected JavaClass(int x)
  {   }
   public void main(String[] args)
   {

    }
}
```

Should the JVM call new JavaClass(int)? What should it pass for x? If not, should the JVM instantiate JavaClass without running any constructor method? because that will special-case your entire class – sometimes you have an instance that hasn't been initialized, and you have to check for it in every method that could be called. There are just too many edge cases and ambiguities for it to make sense for the JVM to have to instantiate a class before the entry point is called. That's why main is static.

**What happens if you remove static modifier from the main method?**
Program compiles successfully . But at runtime throws an error "NoSuchMethodError".

**What is the scope of variables in Java in following cases?**

- **Member Variables** (Class Level Scope) : The member variables must be declared inside class (outside any function). They can be directly accessed anywhere in class
- **Local Variables** (Method Level Scope) : Variables declared inside a method have method level scope and can't be accessed outside the method.
- **Loop Variables** (Block Scope) : A variable declared inside pair of brackets "{" and "}" in a method has scope withing the brackets only.

Read more

**What is "this" keyword in java?**
Within an instance method or a constructor, this is a reference to the current object — the object whose method or constructor is being called. You can refer to any member of the current object from within an instance method or a constructor by using this.
Usage of this keyword

- Used to refer current class instance variable.
- To invoke current class constructor.
- It can be passed as an argument in the method call.
- It can be passed as argument in the constructor call.
- Used to return the current class instance.
- Used to invoke current class method (implicitly)

**What is an abstract class? How abstract classes are similar or different in Java from C++?**
Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods.

- Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.
- Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created
- In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.
- Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

Read more

**Which class is the superclass for every class ?**
Object class

**Can we overload main() method?**
The main method in Java is no extra-terrestrial method. Apart from the fact that main() is just like any other method & can be overloaded in a similar manner, JVM always looks for the method signature to launch the program.

- The normal main method acts as an entry point for the JVM to start the execution of program.
- We can overload the main method in Java. But the program doesn't execute the overloaded main method when we run your program, we need to call the overloaded main method from the actual main method only.

Read more

**What is object cloning?**
Object cloning means to create an exact copy of the original object. If a class needs to support cloning, it must implement java.lang.Cloneable interface and override clone() method from Object class. Syntax of the clone() method is :

```
protected Object clone() throws CloneNotSupportedException
```

If the object's class doesn't implement Cloneable interface then it throws an exception 'CloneNotSupportedException' .

Read more

**How is inheritance in C++ different from Java?**

1. In Java, all classes inherit from the Object class directly or indirectly. Therefore, there is always a single inheritance tree of classes in Java, and Object class is root of the tree.
2. In Java, members of the grandparent class are not directly accessible. See this G-Fact for more details.
3. The meaning of protected member access specifier is somewhat different in Java. In Java, protected members of a class "A" are accessible in other class "B" of same package, even if B doesn't inherit from A (they both have to be in the same package).
4. Java uses *extends* keyword for inheritance. Unlike C++, Java doesn't provide an inheritance specifier like public, protected or private. Therefore, we cannot change the protection level of members of base class in Java, if some data member is public or protected in base class then it remains public or protected in derived class. Like C++, private members of base class are not accessible in derived class.
   Unlike C++, in Java, we don't have to remember those rules of inheritance which are combination of base class access specifier and inheritance specifier.
5. In Java, methods are virtual by default. In C++, we explicitly use virtual keyword. See this G-Fact for more details.
6. Java uses a separate keyword *interface* for interfaces, and *abstract* keyword for abstract classes and abstract functions.
7. Unlike C++, Java doesn't support multiple inheritance. A class cannot inherit from more than one class. A class can implement multiple interfaces though.
8. In C++, default constructor of parent class is automatically called, but if we want to call parametrized constructor of a parent class, we must use Initializer list. Like C++, default constructor of the parent class is automatically called in Java, but if we want to call parameterized constructor then we must use super to call the parent constructor.

See examples here

**Why method overloading is not possible by changing the return type in java?**
In C++ and Java, functions can not be overloaded if they differ only in the return type . The return type of functions is not a part of the mangled name which is generated by the compiler for uniquely identifying each function. The No of arguments, Type of arguments & Sequence of arguments are the parameters which are used to generate the unique mangled name for each function. It is on the basis of these unique mangled names that compiler can understand which function to call even if the names are same(overloading).

**Can we override private methods in Java?**
No, a private method cannot be overridden since it is not visible from any other class. Read more

**What is blank final variable?**
A final variable in Java can be assigned a value only once, we can assign a value either in declaration or later.

```
final int i = 10;
i = 30; // Error because i is final.
```

A **blank final** variable in Java is a final variable that is not initialized during declaration. Below is a simple example of blank final.

```
// A simple blank final example
final int i;
i = 30;
```

Read more

**What is "super" keyword in java?**
The super keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. Whenever you create the instance of subclass, an instance of parent class is created implicitly i.e. referred by super reference variable.
Various scenarios of using java super Keyword:

- super is used to refer immediate parent instance variable
- super is used to call parent class method
- super() is used to call immediate parent constructor

Read more

**What is static variable in Java?**
The static keyword in java is used for memory management mainly. We can apply java static keyword with variables, methods, blocks and nested class. The static keyword belongs to the class than instance of the class.

The static can be:

- variable (also known as class variable)
- method (also known as class method)
- block
- nested class

**Differences between HashMap and HashTable in Java.**
1. HashMap is non synchronized. It is not-thread safe and can't be shared between many threads without proper synchronization code whereas Hashtable is synchronized. It is thread-safe and can be shared with many threads.
2. HashMap allows one null key and multiple null values whereas Hashtable doesn't allow any null key or value.
3. HashMap is generally preferred over HashTable if thread synchronization is not needed
Read more

**How are Java objects stored in memory?**
In Java, all objects are dynamically allocated on **Heap**. This is different from C++ where objects can be allocated memory either on Stack or on Heap. In C++, when we allocate abject using new(), the object is allocated on Heap, otherwise on Stack if not global or static.
In Java, when we only declare a variable of a class type, only a reference is created (memory is not allocated for the object). To allocate memory to an object, we must use new(). So the object is always allocated memory on heap. Read more

**What are C++ features missing in Java?**
Try to answer this on your own before seeing the answer – here.

See also

- Java Multiple Choice Questions
- Practice Coding Questions
- Java articles

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

## GATE CS Notes (According to Official GATE 2017 Syllabus)

## GATE CS Corner

## 10 Most asked Questions from Java Programmers

Hope you liked my previous post "Top 25 Interview Questions". Here comes the next 10.

1) Design discussion on elevator.
Hint: Ask questions related to elevator functionality; come up with a High Level design and Low level design. Be prepared for scheduling questions related to elevator.

2) "n" points are given , find the number of quadruplets which form square.

3) Questions related to memory management in Java.

4) Mark and sweep algorithm and garbage collection in Java

5) Construct tree from Inorder and Preorder

6) Serialization in Java

7) How to ensure that instance is never garbage collected?
Hint: We can use singleton pattern. There's a static reference to a singleton, so it won't be eligible for garbage collection until the classloader is eligible for garbage collection.

8) Questions related to classloader, rt.jar?

9) Difference between String, StringBuffer and StringBuilder?

10) Why String is immutable in Java?
Note: Do discuss about security related issues in class loader.

- Read More- Java Articles
- MCQ Quizzes of Java

Thanks to GeeksforGeeks team for providing a nice platform. You guys are the best.

This article is contributed by Rishi Verma. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

**GATE CS Corner    Company Wise Coding Practice**

Interview Experiences
Interview Tips
placement preperation

## Google Interview Experience | Set 5 (for Java Position)

The solution will be evaluated on following parameters.
Object Oriented Design aspects of the solution.
Overall coding practices.
Working test cases of the solution.

You can use Ant/Maven as build tools for the solution, Junit, Mockito or other testing frameworks.
You may also include a brief explanation of your design and assumptions along with your code.

***Problem Statement:*** In a Formula-1 challenge, there are n teams numbered 1 to n. Each team has a car and a driver. Car's specification are as follows:
– Top speed: (150 + 10 * i) km per hour
– Acceleration: (2 * i) meter per second square.
– Handling factor (hf) = 0.8
– Nitro : Increases the speed to double or top speed, whichever is less. Can be used only once.

Here i is the team number.
The cars line up for the race. The start line for (i + 1)th car is 200 * i meters behind the ith car.

All of them start at the same time and try to attain their top speed. A re-assessment of the positions is done every 2 seconds(So even if the car has crossed the finish line in between, you'll get to know after 2 seconds). During this assessment, each driver checks if there is any car within 10 meters of his car, his speed reduces to: hf * (speed at that moment). Also, if the driver notices that he is the last one on the race, he uses 'nitro'.

Taking the number of teams and length of track as the input, Calculate the final speeds and the corresponding completion times.

If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

**GATE CS Corner    Company Wise Coding Practice**

Interview Experiences
Google

Question 1

```
class Main {
   public static void main(String args[]) {
        int t;
        System.out.println(t);
   }
}
```

A0
Bgarbage value
Ccompiler error
Druntime error

**Data Types**
**Discuss it**
Question 1 Explanation:
Unlike class members, local variables of methods must be assigned a value to before they are accessed, or it is a compile error.
Question 2
Predict the output of following Java program.

```
class Test {
    public static void main(String[] args) {
      for(int i = 0; 0; i++)
      {
          System.out.println("Hello");
          break;
      }
    }
}
```

AHello
BEmpty Output
CCompiler error
DRuntime error
**Data Types**
**Discuss it**
Question 2 Explanation:
The error is in for loop where 0 is used in place of boolean value. Unlike C++, use of non boolean variables in place of bool is not allowed
Question 3
Predict the output of the following program.

```
 class Test
{
    public static void main(String[] args)
    {
        Double object = new Double("2.4");
        int a = object.intValue();
        byte b = object.byteValue();
        float d = object.floatValue();
        double c = object.doubleValue();

        System.out.println(a + b + c + d );

    }
}
```

A8
B8.8
C8.800000095367432
**Data Types**
**Discuss it**
Question 3 Explanation:
Arithmetic conversions are implicitly performed to cast the values to a common type. The compiler first performs integer promotion. If the operands still have different types, then they are converted to the type that appears highest in the hierarchy.
There are 3 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

---

Question 1
Predict the output of following Java Program

```
class Test {
    public static void main(String args[])  {
       int x = -4;
       System.out.println(x>>1);
       int y = 4;
       System.out.println(y>>1);
    }
}
```

ACompiler Error: Operator >> cannot be applied to negative numbers

B
```
-2
2
```

C
```
2
2
```

D
```
0
2
```

**Operators**
**Discuss it**
Question 1 Explanation:
See http://www.geeksforgeeks.org/bitwise-shift-operators-in-java/
Question 2
Predict the output of following Java program. Assume that int is stored using 32 bits.

```
class Test {
    public static void main(String args[])  {
       int x = -1;
       System.out.println(x>>>29);
       System.out.println(x>>>30);
       System.out.println(x>>>31);
   }
}
```

A
```
7
3
1
```

B
```
15
7
3
```

C
```
0
0
0
```

D
```
1
1
1
```

**Operators**
**Discuss it**
Question 2 Explanation:
Please see http://www.geeksforgeeks.org/bitwise-shift-operators-in-java/
Question 3

```
class Test {
    public static void main(String args[])  {
       System.out.println(10  +  20 + "GeeksQuiz");
       System.out.println("GeeksQuiz" + 10 + 20);
   }
}
```

A
```
30GeeksQuiz
GeeksQuiz30
```

B
```
1020GeeksQuiz
GeeksQuiz1020
```

C
```
30GeeksQuiz
GeeksQuiz1020
```

D
```
1020GeeksQuiz
GeeksQuiz30
```

**Operators**
**Discuss it**
Question 3 Explanation:
In the given expressions **10 + 20 + "GeeksQuiz"** and **"GeeksQuiz" + 10 + 20** , there are two + operators, so associativity comes to the picture. The + operator is left to right. So the first expression is evaluated as **(10 + 20) + "GeeksQuiz"** and second expression is evaluated as **("GeeksQuiz" + 10) + 20** .
Question 4

```
class Test {
    public static void main(String args[])  {
       System.out.println(10*20 + "GeeksQuiz");
       System.out.println("GeeksQuiz" + 10*20);
   }
}
```

A
```
10*20GeeksQuiz
GeeksQuiz10*20
```

B
```
200GeeksQuiz
GeeksQuiz200
```

C
```
200GeeksQuiz
GeeksQuiz10*20
```

D
```
1020GeeksQuiz
GeeksQuiz220
```

**Operators**
**Discuss it**
Question 4 Explanation:
Precedence of * is more than +.
Question 5
Which of the following is not an operator in Java?
Ainstanceof
Bsizeof
Cnew
D>>>=
**Operators**
**Discuss it**
Question 5 Explanation:
There is no sizeof operator in Java. We generally don't need size of objects.
Question 6

```
class Base {}

class Derived extends Base {
   public static void main(String args[]){
      Base a = new Derived();
      System.out.println(a instanceof Derived);
   }
}
```

Atrue
Bfalse
**Operators**
**Discuss it**
Question 6 Explanation:
The instanceof operator works even when the reference is of base class type.
Question 7

```
class Test
{
    public static void main(String args[])
    {
        String s1 = "geeksquiz";
        String s2 = "geeksquiz";
        System.out.println("s1 == s2 is:" + s1 == s2);
    }
}
```

Atrue
Bfalse
Ccompiler error
Dthrows an exception
**Operators**
**Discuss it**
Question 7 Explanation:
The output is “false” because in java + operator precedence is more than == operator. So the given expression will be evaluated to “**s1 == s2 is:geeksquiz**” == “**geeksquiz**” i.e false.
Question 8

```
class demo
{
    int a, b, c;
    demo(int a, int b, int c)
    {
        this.a = a;
        this.b = b;
    }
    demo()
    {
        a = b = c = 0;
    }

    demo operator+(const demo &obj)
    {
        demo object;
        object.a = this.a + obj.a;
        object.b = this.b + obj.b;
        object.c = this.c + obj.c;
        return object;
    }
}

class Test
{
    public static void main(String[] args)
    {

        demo obj1 = new demo(1, 2, 3);
        demo obj2 = new demo(1, 2, 3);
        demo obj3 = new demo();

        obj3 = obj1 + obj2;
        System.out.println ("obj3.a = " + obj3.a);
        System.out.println ("obj3.b = " + obj3.c);
        System.out.println ("obj3.c = " + obj3.c);

    }
}
```

ACompile Error
BRun Time Error
CSegmentation Fault
**Operators**
**Discuss it**
Question 8 Explanation:
Operator overloading is not support by JAVA. It only supports method overloading, whereas C++ supports both method and operator overloading.
Question 9
Predict the output of the following program.

```
 class Test
{
    boolean[] array = new boolean[3];
    int count = 0;

    void set(boolean[] arr, int x)
    {
        arr[x] = true;
        count++;
    }

    void func()
    {
        if(array[0] && array[++count - 2] | array [count - 1])
            count++;

        System.out.println("count = " + count);
    }


    public static void main(String[] args)
    {
        Test object = new Test();
        object.set(object.array, 0);
        object.set(object.array, 1);
        object.func();
    }
}
```

A2
B3
C4
**Operators**
**Discuss it**
Question 9 Explanation:

First call to function set(), sets array[0] = true, array[1] = false and array[2] = false. Second call to function set(), sets array[0] = true, array[1] = true and array[2] = false. In function func(),if statement evaluates to be true. So, count = 4.
There are 9 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the following is FALSE about arrays on Java
AA java array is always an object
BLength of array can be changed after creation of array
CArrays in Java are always allocated on heap
**Arrays**
**Discuss it**
Question 1 Explanation:
In Java, arrays are objects, they have members like length. The length member is final and cannot be changed. All objects are allocated on heap in Java, so arrays are also allocated on heap.
Question 2
Predict the output?

```
// file name: Main.java
public class Main {
    public static void main(String args[]) {
       int arr[] = {10, 20, 30, 40, 50};
       for(int i=0; i < arr.length; i++)
```

```
        {
            System.out.print(" " + arr[i]);
        }
    }
}
```

A10 20 30 40 50
BCompiler Error
C10 20 30 40
**Arrays**
**Discuss it**
Question 2 Explanation:
It is a simple program where an array is first created then traversed. The important thing to note is, unlike C++, arrays are first class objects in Java. For example, in the following program, size of array is accessed using length which is a member of arr[] object.
Question 3

```
class Test {
   public static void main(String args[]) {
     int arr[2];
     System.out.println(arr[0]);
     System.out.println(arr[1]);
   }
}
```

A
```
0
0
```

B
```
garbage value
garbage value
```

CCompiler Error
DException
**Arrays**
**Discuss it**
Question 3 Explanation:
In Java, it is not allowed to put the size of the array in the declaration because an array declaration specifies only the element type and the variable name. The size is specified when you allocate space for the array. Even the following simple program won't compile.

```
class Test {
   public static void main(String args[]) {
     int arr[5];   //Error
   }
}
```

Question 4

```
class Test {
   public static void main(String args[]) {
     int arr[] = new int[2];
     System.out.println(arr[0]);
     System.out.println(arr[1]);
   }
}
```

A
```
0
0
```

B
```
garbage value
garbage value
```

CCompiler Error
DException
**Arrays**
**Discuss it**
Question 4 Explanation:
Java arrays are first class objects and all members of objects are initialized with default values like o, null.
Question 5

```
public class Main {
    public static void main(String args[]) {
        int arr[][] = new int[4][];
        arr[0] = new int[1];
        arr[1] = new int[2];
        arr[2] = new int[3];
        arr[3] = new int[4];

        int i, j, k = 0;
        for (i = 0; i < 4; i++) {
            for (j = 0; j < i + 1; j++) {
                arr[i][j] = k;
                k++;
            }
        }
        for (i = 0; i < 4; i++) {
            for (j = 0; j < i + 1; j++) {
                System.out.print(" " + arr[i][j]);
                k++;
            }
            System.out.println();
        }
    }
}
```

ACompiler Error
B
```
0
1 2
3 4 5
6 7 8 9
```

C
```
0
0 0
0 0 0
0 0 0 0
```

D
```
9
7 8
4 5 6
```

```
0 1 2 3
```

**Arrays**
**Discuss it**
Question 5 Explanation:
In Java, we can create jagged arrays.  Refer Jagged Array in Java for details.
Question 6
Output of following Java program?

```
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (arr1 == arr2)
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

ASame
BNot Same
**Arrays**
**Discuss it**
Question 6 Explanation:
See http://www.geeksforgeeks.org/compare-two-arrays-java/
Question 7
Output of following Java program?

```
import java.util.Arrays;
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (Arrays.equals(arr1, arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

ASame
BNot Same
**Arrays**
**Discuss it**
Question 7 Explanation:
See http://www.geeksforgeeks.org/compare-two-arrays-java/
Question 8

```
class Test
{
    public static void main (String[] args)
    {
        int arr1[] = {1, 2, 3};
        int arr2[] = {1, 2, 3};
        if (arr1.equals(arr2))
            System.out.println("Same");
        else
            System.out.println("Not same");
    }
}
```

ASame
BNot same
**Arrays**
**Discuss it**
Question 8 Explanation:
arr1.equals(arr2) is same as (arr1 == arr2)
Question 9
Consider the following C program which is supposed to compute the transpose of a given 4 x 4 matrix M. Note that, there is an X in the program which indicates some missing statements. Choose the correct option to replace X in the program.

```
#include<stdio.h>
#define ROW 4
#define COL 4
int M[ROW][COL] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16};
main()
{
    int i, j, t;
    for (i = 0; i < 4; ++i)
    {
        X
    }
    for (1 = 0; i < 4; ++i)
        for (j = 0; j < 4; ++j)
            printf ("%d", M[i][j]);
}
```

A)
```
for(j = 0; j < 4; ++j){
     t = M[i][j];
     M[i][j] = M[j][i];
     M[j][i] = t;
}
```

B)
```
for(j = 0; j < 4; ++j){
     M[i][j] = t;
     t = M[j][i];
     M[j][i] = M[i][j];
}
```

C)
```
for(j = i; j < 4; ++j){
     t = M[i][j];
     M[i][j] = M[j][i];
```

```
    M[j][i] = t;
}
```

D)
```
for(j = i; j < 4; ++j){
    M[i][j] = t;
    t = M[j][i];
    M[j][i] = M[i][j];
}
```

AA
BB
CC
DD
**Arrays** **GATE-IT-2004**
**Discuss it**
Question 9 Explanation:

- To compute transpose **j needs to be started with i**,so A and B are WRONG
- In D, given statement is wrong as temporary variable t needs to be assigned some value and NOT vice versa

```
M[i][j] = t;
```

**So the answer is C** Check out the correct option C at Solution: http://code.geeksforgeeks.org/r7wbP6
There are 9 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

---

Question 1
Output of following Java program?

```
class Main {
    public static void main(String args[]) {
            System.out.println(fun());
    }

    int fun()
    {
      return 20;
    }
}
```

A20
Bcompiler error
C0
Dgarbage balue
**Functions**
**Discuss it**
Question 1 Explanation:
main() is a static method and fun() is a non-static method in class Main. Like C++, in Java calling a non-static function inside a static function is not allowed
Question 2

```
public class Main {
    public static void main(String args[]) {
       String x = null;
       giveMeAString(x);
       System.out.println(x);
    }
    static void giveMeAString(String y)
    {
       y = "GeeksQuiz";
    }
}
```

AGeeksQuiz
Bnull
CCompiler Error
DException
**Functions**
**Discuss it**
Question 2 Explanation:
Parameters in Java is passed by value. So the changes made to y do not reflect in main().
Question 3

```
class Test {
public static void swap(Integer i, Integer j) {
      Integer temp = new Integer(i);
      i = j;
      j = temp;
   }
   public static void main(String[] args) {
      Integer i = new Integer(10);
      Integer j = new Integer(20);
      swap(i, j);
      System.out.println("i = " + i + ", j = " + j);
   }
}
```

Ai = 10, j = 20
Bi = 20, j = 10
Ci = 10, j = 10
Di = 20, j = 20
**Functions**
**Discuss it**
Question 3 Explanation:
Parameters are passed by value in Java
Question 4

```
class intWrap {
   int x;
}
public class Main {
    public static void main(String[] args) {
       intWrap i = new intWrap();
       i.x = 10;
       intWrap j = new intWrap();
       j.x = 20;
       swap(i, j);
       System.out.println("i.x = " + i.x + ", j.x = " + j.x);
    }
    public static void swap(intWrap i, intWrap j) {
       int temp = i.x;
       i.x = j.x;
       j.x = temp;
    }
}
```

Ai.x = 20, j.x = 10
Bi.x = 10, j.x = 20
Ci.x = 10, j.x = 10
Di.x = 20, j.x = 20
**Functions**
**Discuss it**
Question 4 Explanation:
Objects are never passed at all. Only references are passed. The values of variables are always primitives or references, never objects
Question 5
In Java, can we make functions inline like C++?
AYes
BNo
**Functions**
**Discuss it**
Question 5 Explanation:
Unlike C++, in Java we can't suggest functions as inline. Java compiler does some complicated analysis and may make functions inline internally.
Question 6
Predict the output?

```
class Main {
    public static void main(String args[]) {
            System.out.println(fun());
    }
    static int fun(int x = 0)
    {
      return x;
    }
}
```

A0
BGarbage Value
CCompiler Error
DRuntime Error
**Functions**
**Discuss it**
Question 6 Explanation:
Java doesn't support default arguments. In Java, we must write two different functions.
Question 7
Predict the output of the following program.

```
class Test
{
    public void demo(String str)
    {
        String[] arr = str.split(";");
        for (String s : arr)
        {
            System.out.println(s);
        }
    }

    public static void main(String[] args)
    {
        char array[] = {'a', 'b', ' ', 'c', 'd', ';', 'e', 'f', ' ',
                        'g', 'h', ';', 'i', 'j', ' ', 'k', 'l'};
        String str = new String(array);
        Test obj = new Test();
        obj.demo(str);
    }
}
```

A
```
ab cd
ef gh
ij kl
```

B
```
ab
cd;ef
gh;ij
kl
```

CCompilation error
**Functions**
**Discuss it**
Question 7 Explanation:

Class String has a inbuilt parameterized constructor **String(character_array)** which initializes string 'str' with the values stored in character array. The split() method splits the string based on the given regular expression or delimiter passed it as parameter and returns an array.
Question 8
Predict the output of the following program.

```

class Test
{
    public static void main(String[] args)
    {
        StringBuffer a = new StringBuffer("geeks");
        StringBuffer b = new StringBuffer("forgeeks");
        a.delete(1,3);
        a.append(b);
        System.out.println(a);
    }
}
```

Agsforgeeks

Bgksforgeeks
Cgeksforgeeks
DCompilation error
**Functions**
**Discuss it**
Question 8 Explanation:

delete(x, y) function deletes the elements from the string at position ‘x’ to position ‘y-1’. append() function concatenates the second string to the first string.
Question 9
Predict the output of the following program.

```
 class Test
{

    public static void main(String[] args)
    {
        String obj1 = new String("geeks");
        String obj2 = new String("geeks");

        if(obj1.hashCode() == obj2.hashCode())
            System.out.println("hashCode of object1 is equal to object2");

        if(obj1 == obj2)
            System.out.println("memory address of object1 is same as object2");

        if(obj1.equals(obj2))
            System.out.println("value of object1 is equal to object2");
    }
}
```

A
```
hashCode of object1 is equal to object2
value of object1 is equal to object2
```

B
```
hashCode of object1 is equal to object2
memory address of object1 is same as object2
value of object1 is equal to object2
```

C
```
memory address of object1 is same as object2
value of object1 is equal to object2
```

**Functions**
**Discuss it**
Question 9 Explanation:

obj.hashCode() function returns a 32-bit hash code value for the object ‘obj’. obj1.equals(obj2) function returns true if value of obj1 is equal to obj2. obj1 == obj2 returns true if obj1 refers to same memory address as obj2.
Question 10
Predict the output of the following program.

```
 class Test implements Cloneable
{
    int a;

    Test cloning()
    {
        try
        {
            return (Test) super.clone();
        }
        catch(CloneNotSupportedException e)
        {
            System.out.println("CloneNotSupportedException is caught");
            return this;
        }
    }
}

class demo
{

    public static void main(String args[])
    {
        Test obj1 = new Test();
        Test obj2;
        obj1.a = 10;
        obj2 = obj1.cloning();
        obj2.a = 20;

        System.out.println("obj1.a = " + obj1.a);
        System.out.println("obj2.a = " + obj2.a);
    }
}
```

A
```
obj1.a = 10
obj2.a = 20
```

B
```
obj1.a = 20
obj2.a = 20
```

C
```
obj1.a = 10
obj2.a = 10
```

**Functions**
**Discuss it**
Question 10 Explanation:
The clone( ) method generates a duplicate copy of the object on which it is called. Only classes that implement the Cloneable interface can be cloned.
Question 11
Predict the output of the following program.

```
class Test
{
    public static void main(String[] args)
    {
        String str = "geeks";
        str.toUpperCase();
        str += "forgeeks";
        String string = str.substring(2,13);
        string = string + str.charAt(4);;
        System.out.println(string);
    }
}
```

Aeksforgeekss
Beksforgeeks
CEKSforgeekss
DEKSforgeeks
**Functions**
**Discuss it**
Question 11 Explanation:
str.toUpperCase() returns 'str' in upper case. But,it does not change the original string 'str'. str.substring(x, y) returns a string from position 'x'(inclusive) to position 'y'(exclusive). str.charAt(x) returns a character at postion 'x' in the string 'str'.
Question 12
The function f is defined as follows:

```
int f (int n) {
    if (n <= 1) return 1;
    else if (n % 2  ==  0) return f(n/2);
    else return f(3n - 1);
}
```

Assuming that arbitrarily large integers can be passed as a parameter to the function, consider the following statements.
1. The function f terminates for finitely many different values of n ≥ 1.
ii. The function f terminates for infinitely many different values of n ≥ 1.
iii. The function f does not terminate for finitely many different values of n ≥ 1.
iv. The function f does not terminate for infinitely many different values of n ≥ 1.
Which one of the following options is true of the above?
A(i) and (iii)
B(i) and (iv)
C(ii) and (iii)
D(ii) and (iv)
**Recursion**    **Functions**    **C Quiz - 113**    **Gate IT 2007**
**Discuss it**
Question 12 Explanation:
The function terminates for all values having a factor of 2 {(2.x)2==0}
So, (i) is false and (ii) is TRUE.
Let n = 3, it will terminate in 2nd iteration.
Let n=5, it will go like 5 - 14 - 7 - 20 - 10 - 5 – and now it will repeat.
And any number with a factor of 5 and 2, there are infinite recursions possible.
So, (iv) is TRUE and (iii) is false.
There are 12 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Predict the output of following Java program?

```
class Test {
  int i;
}
class Main {
   public static void main(String args[]) {
     Test t;
     System.out.println(t.i);
}
```

A0
Bgarbage value
Ccompiler error
Druntime error
**Class and Object**
**Discuss it**
Question 1 Explanation:
t is just a reference, the object referred by t is not allocated any memory. Unlike C++, in Java all non-primitive objects must be explicitly allocated and these objects are allocated on heap. The following is corrected program. 1
Question 2
Predict the output of following Java program

```
class Test {
  int i;
}
class Main {
  public static void main(String args[]) {
      Test t = new Test();
      System.out.println(t.i);
   }
}
```

Agarbage value
B0
Ccompiler error
Druntime error
**Class and Object**
**Discuss it**
Question 2 Explanation:
In Java, fields of classes and objects that do not have an explicit initializer and elements of arrays are automatically initialized with the default value for their type (false for boolean, 0 for all numerical types, null for all reference types). Local variables in Java must be definitely assigned to before they are accessed, or it is a compile error.
Question 3

```
class demo
{
    int a, b;

    demo()
    {
        a = 10;
        b = 20;
    }

    public void print()
    {
        System.out.println ("a = " + a + " b = " + b + "\n");
    }
}

class Test
{

    public static void main(String[] args)
    {
        demo obj1 = new demo();
        demo obj2 = obj1;

        obj1.a += 1;
        obj1.b += 1;

        System.out.println ("values of obj1 : ");
        obj1.print();
        System.out.println ("values of obj2 : ");
        obj2.print();

    }
}
```

ACompile error

B
```
values of obj1:
a = 11 b = 21
values of obj2:
a = 11 b = 21
```

C
```
values of obj1:
a = 11 b = 21
values of obj2:
a = 10 b = 20
```

D
```
values of obj1:
a = 11 b = 20
values of obj2:
a = 10 b = 21
```

ERun time error

**Class and Object**

**Discuss it**

Question 3 Explanation:

Assignment of obj2 to obj1 makes obj2 a reference to obj1. Therefore, any change in obj1 will be reflected in obj2 also.

Question 4

Predict the output of following Java program.

```
 class demoClass
{
    int a = 1;

    void func()
    {
        demo obj = new demo();
        obj.display();
    }


    class demo
    {
        int b = 2;

        void display()
        {
            System.out.println("\na = " + a);
        }
    }

    void get()
    {
        System.out.println("\nb = " + b);
    }
}


class Test
{
    public static void main(String[] args)
    {
        demoClass obj = new demoClass();
        obj.func();
        obj.get();

    }
}
```

A
```
a = 1
b = 2
```

BCompilation error

C
```
b = 2
a = 1
```

**Class and Object**

**Discuss it**

Question 4 Explanation:

Members of inner class ‘demo’ can not be used in the outer class ‘Test’. Thus, get() of outer class can not access variable ‘b’ of inner class.

Question 5

Predict the output of the following program.

```
class First
{
    void display()
    {
        System.out.println("Inside First");
    }
}

class Second extends First
{

    void display()
    {
        System.out.println("Inside Second");
    }
}


class Test
{

    public static void main(String[] args)
    {
        First obj1 =  new First();
        Second obj2 =  new Second();

        First ref;
        ref = obj1;
        ref.display();

        ref = obj2;
        ref.display();
    }
}
```

ACompilation error

B
```
Inside First
Inside Second
```

C
```
Inside First
Inside First
```

DRuntime error

**Class and Object**

**Discuss it**

Question 5 Explanation:

‘ref’ is a reference variable which obtains the reference of object of class First and calls its function display(). Then ‘ref’ refers to object of class Second and calls its function display().

Question 6

Predict the output of the following program.

```
 class Test
{
    int a = 1;
    int b = 2;

    Test func(Test obj)
    {
        Test obj3 = new Test();
        obj3 = obj;
        obj3.a = obj.a++ + ++obj.b;
        obj.b = obj.b;
        return obj3;
    }

    public static void main(String[] args)
    {
        Test obj1 = new Test();
        Test obj2 = obj1.func(obj1);

        System.out.println("obj1.a = " + obj1.a + "  obj1.b = " + obj1.b);
        System.out.println("obj2.a = " + obj2.a + "  obj1.b = " + obj2.b);

    }
}
```

A
```
obj1.a = 1  obj1.b = 2
obj2.a = 4  obj2.b = 3
```

B
```
obj1.a = 4  obj1.b = 3
obj2.a = 4  obj2.b = 3
```

CCompilation error

**Class and Object**

**Discuss it**

Question 6 Explanation:

obj1 and obj2 refer to same memory address.

Question 7

Predict the output of the following program.

```
 class Test
{
    int a = 1;
    int b = 2;

    Test func(Test obj)
    {
        Test obj3 = new Test();
        obj3 = obj;
        obj3.a = obj.a++ + ++obj.b;
        obj.b = obj.b;
        return obj3;
    }

    public static void main(String[] args)
    {
        Test obj1 = new Test();
        Test obj2 = obj1.func(obj1);
        System.out.println("obj1.a = " + obj1.a + "  obj1.b = " + obj1.b);
        System.out.println("obj2.a = " + obj2.a + "  obj1.b = " + obj2.b);

    }
}
```

A
```
obj1.a = 1  obj1.b = 2
obj2.a = 4  obj2.b = 3
```

B
```
obj1.a = 4  obj1.b = 3
obj2.a = 4  obj2.b = 3
```

CCompilation error
**Class and Object**
**Discuss it**
Question 7 Explanation:

obj1 and obj2 refer to same memory address.
There are 7 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Predict the output?

```
package main;
class T {
  int t = 20;
}
class Main {
   public static void main(String args[]) {
      T t1 = new T();
      System.out.println(t1.t);
   }
}
```

A20
B0
CCompiler Error
**Constructors**
**Discuss it**
Question 1 Explanation:
In Java, member variables can assigned a value with declaration. In C++, only static const variables can be assigned like this.
Question 2
Predict the output of following Java program

```
class T {
  int t = 20;
  T() {
    t = 40;
  }
}
class Main {
   public static void main(String args[]) {
      T t1 = new T();
      System.out.println(t1.t);
   }
}
```

A20
B40
CCompiler Error
**Constructors**
**Discuss it**
Question 2 Explanation:
The values assigned inside constructor overwrite the values initialized with declaration.
Question 3
Which of the following is/are true about constructors in Java?

```

1) Constructor name should be same as class name.
2) If you don't define a constructor for a class,
    a default parameterless constructor is automatically
    created by the compiler.
3) The default constructor calls super() and initializes all
   instance variables to default value like 0, null.
4) If we want to parent class constructor, it must be called in
   first line of constructor.
```

A1
B1, 2
C1, 2 and 3
D1, 2, 3 and 4
**Constructors**
**Discuss it**
Question 4
Is there any compiler error in the below Java program?

```
class Point {
    int m_x, m_y;
    public Point(int x, int y) {    m_x = x;    m_y = y;  }
    public static void main(String args[])
    {
      Point p = new Point();
    }
}
```

AYes
BNo
**Constructors**

**Discuss it**
Question 4 Explanation:
The main function calls paramaterless constructor, but there is only one constructor defined in class which takes two parameters. Note that if we write our own constructor, then compiler doesn't create default constructor in Java. This behavior is same as C++.
Question 5
Output of following Java program

```
class Point {
  int m_x, m_y;

  public Point(int x, int y) { m_x = x; m_y = y; }
  public Point() { this(10, 10); }
  public int getX() { return m_x; }
  public int getY() { return m_y; }

  public static void main(String args[]) {
    Point p = new Point();
    System.out.println(p.getX());
  }
}
```

A10
B0
Ccompiler error
**Constructors**
**Discuss it**
Question 5 Explanation:
it's a simple program where constructor is called with parameters and values are initialized.
Question 6

```
final class Complex {
    private  double re,  im;
    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }
    Complex(Complex c)
    {
      System.out.println("Copy constructor called");
      re = c.re;
      im = c.im;
    }
    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}
class Main {
    public static void main(String[] args) {
        Complex c1 = new Complex(10, 15);
        Complex c2 = new Complex(c1);
        Complex c3 = c1;
        System.out.println(c2);
    }
}
```

A
```
Copy constructor called
(10.0 + 15.0i)
```

B
```
Copy constructor called
(0.0 + 0.0i)
```

C
```
(10.0 + 15.0i)
```

D
```
(0.0 + 0.0i)
```

**Constructors**
**Discuss it**
Question 7

```
class Test
{
    static int a;

    static
    {
        a = 4;
        System.out.println ("inside static block\n");
        System.out.println ("a = " + a);
    }

    Test()
    {
        System.out.println ("\ninside constructor\n");
        a = 10;
    }

    public static void func()
    {
        a = a + 1;
        System.out.println ("a = " + a);
    }

    public static void main(String[] args)
    {

        Test obj = new Test();
        obj.func();

    }
}
```

A
```
inside static block
a = 4
inside constructor
a = 11
```

BCompiler Error
CRun Time Error

D
```
inside static block
a = 4
inside constructor
a = 5
```

E
```
inside static block
a = 10
inside constructor
a = 11
```

**Constructors**
**Discuss it**
Question 7 Explanation:
Static blocks are called before constructors. Therefore, on object creation of class Test, static block is called. So, static variable a = 4. Then constructor Test() is called which assigns a = 10. Finally, function func() increments its value.
There are 7 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

---

Question 1
Output of following Java Program?

```
class Base {
    public void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
}

public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

ADerived::show() called
BBase::show() called
**Inheritance**
**Discuss it**
Question 1 Explanation:
In the above program, b is a reference of Base type and refers to an abject of Derived class. In Java, functions are virtual by default. So the run time polymorphism happens and derived fun() is called.
Question 2

```
class Base {
    final public void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
}

class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

ABase::show() called
BDerived::show() called
CCompiler Error
DRuntime Error
**Inheritance**
**Discuss it**
Question 2 Explanation:
Final methods cannot be overridden. See the compiler error here.
Question 3

```
class Base {
    public static void show() {
       System.out.println("Base::show() called");
    }
}

class Derived extends Base {
    public static void show() {
       System.out.println("Derived::show() called");
    }
}

class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

ABase::show() called
BDerived::show() called
CCompiler Error
**Inheritance**
**Discuss it**
Question 3 Explanation:

Like C++, when a function is static, runtime polymorphism doesn't happen.
Question 4
Which of the following is true about inheritance in Java?

```
1) Private methods are final.
2) Protected members are accessible within a package and
   inherited classes outside the package.
3) Protected methods are final.
4) We cannot override private methods.
```

A1, 2 and 4
BOnly 1 and 2
C1, 2 and 3
D2, 3 and 4
**Inheritance**
**Discuss it**
Question 4 Explanation:
See http://www.geeksforgeeks.org/can-override-private-methods-java/ and http://www.geeksforgeeks.org/comparison-of-inheritance-in-c-and-java/
Question 5
Output of following Java program?

```
class Base {
    public void Print() {
        System.out.println("Base");
    }
}

class Derived extends Base {
    public void Print() {
        System.out.println("Derived");
    }
}

class Main{
    public static void DoPrint( Base o ) {
        o.Print();
    }
    public static void main(String[] args) {
        Base x = new Base();
        Base y = new Derived();
        Derived z = new Derived();
        DoPrint(x);
        DoPrint(y);
        DoPrint(z);
    }
}
```

A
```
Base
Derived
Derived
```

B
```
Base
Base
Derived
```

C
```
Base
Derived
Base
```

DCompiler Error
**Inheritance**
**Discuss it**
Question 5 Explanation:
See question 1 of http://www.geeksforgeeks.org/output-of-java-program-set-2/
Question 6
Predict the output of following program. Note that fun() is public in base and private in derived.

```
class Base {
    public void foo() { System.out.println("Base"); }
}

class Derived extends Base {
    private void foo() { System.out.println("Derived"); }
}

public class Main {
    public static void main(String args[]) {
        Base b = new Derived();
        b.foo();
    }
}
```

ABase
BDerived
CCompiler Error
DRuntime Error
**Inheritance**
**Discuss it**
Question 6 Explanation:
It is compiler error to give more restrictive access to a derived class function which overrides a base class function.
Question 7
Which of the following is true about inheritance in Java. 1) In Java all classes inherit from the Object class directly or indirectly. The Object class is root of all classes. 2) Multiple inheritance is not allowed in Java. 3) Unlike C++, there is nothing like type of inheritance in Java where we can specify whether the inheritance is protected, public or private.
A1, 2 and 3
B1 and 2
C2 and 3
D1 and 3
**Inheritance**
**Discuss it**
Question 7 Explanation:
See Comparison of Inheritance in C++ and Java
Question 8
Predict the output of following Java Program

```
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.super.Print();
        System.out.println("Child's Print()");
    }
}

public class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

ACompiler Error in super.super.Print()

B
```
Grandparent's Print()
Parent's Print()
Child's Print()
```

CRuntime Error

**Inheritance**

**Discuss it**

Question 8 Explanation:

In Java, it is not allowed to do super.super. We can only access Grandparent's members using Parent. For example, the following program works fine.

```
// Guess the output
// filename Main.java
class Grandparent {
    public void Print() {
        System.out.println("Grandparent's Print()");
    }
}

class Parent extends Grandparent {
    public void Print() {
     super.Print();
        System.out.println("Parent's Print()");
    }
}

class Child extends Parent {
    public void Print() {
        super.Print();
        System.out.println("Child's Print()");
    }
}

class Main {
    public static void main(String[] args) {
        Child c = new Child();
        c.Print();
    }
}
```

Question 9

```
final class Complex {

    private final double re;
    private final double im;

    public Complex(double re, double im) {
        this.re = re;
        this.im = im;
    }

    public String toString() {
        return "(" + re + " + " + im + "i)";
    }
}

class Main {
  public static void main(String args[])
  {
       Complex c = new Complex(10, 15);
       System.out.println("Complex number is " + c);
  }
}
```

A
```
Complex number is (10.0 + 15.0i)
```

BCompiler Error

C
```
Complex number is SOME_GARBAGE
```

D
```
Complex number is Complex@8e2fb5
```

Here 8e2fb5 is hash code of c

**Inheritance**

**Discuss it**

Question 9 Explanation:

See http://www.geeksforgeeks.org/overriding-tostring-method-in-java/

There are 9 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

Question 1
Which of the following is FALSE about abstract classes in Java
AIf we derive an abstract class and do not implement all the abstract methods, then the derived class should also be marked as abstract using 'abstract' keyword
BAbstract classes can have constructors
CA class can be made abstract without any abstract method
DA class can inherit from multiple abstract classes.
**Abstract Class and Interface**
**Discuss it**
Question 2
Which of the following is true about interfaces in java.

```
1) An interface can contain following type of members.
....public, static, final fields (i.e., constants)
....default and static methods with bodies

2) An instance of interface can be created.

3) A class can implement multiple interfaces.

4) Many classes can implement the same interface.
```

A1, 3 and 4
B1, 2 and 4
C2, 3 and 4
D1, 2, 3 and 4
**Abstract Class and Interface**
**Discuss it**
Question 3
Predict the output of the following program.

```
abstract class demo
{
    public int a;
    demo()
    {
        a = 10;
    }

    abstract public void set();

    abstract final public void get();

}

class Test extends demo
{

    public void set(int a)
    {
        this.a = a;
    }

    final public void get()
    {
        System.out.println("a = " + a);
    }

    public static void main(String[] args)
    {
        Test obj = new Test();
        obj.set(20);
        obj.get();
    }
}
```

Aa = 10
Ba = 20
CCompilation error
**Abstract Class and Interface**
**Discuss it**
Question 3 Explanation:
Final method can’t be overridden. Thus, an abstract function can’t be final.
There are 3 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

---

Question 1
What is the use of final keyword in Java?
AWhen a class is made final, a sublcass of it can not be created.
BWhen a method is final, it can not be overridden.
CWhen a variable is final, it can be assigned value only once.
DAll of the above
**final keyword**
**Discuss it**
Question 1 Explanation:
See final in Java.
Question 2
Output of follwoing Java program

```
class Main {
 public static void main(String args[]){
   final int i;
   i = 20;
   System.out.println(i);
 }
}
```

A20
BCompiler Error
C0

DGarbage value
**final keyword**
**Discuss it**
Question 2 Explanation:
There is no error in the program. final variables can be assigned value only once. In the above program, i is assigned a value as 20, so 20 is printed.
Question 3

```
class Main {
 public static void main(String args[]){
    final int i;
    i = 20;
    i = 30;
    System.out.println(i);
 }
}
```

A30
BCompiler Error
CGarbage value
D0
**final keyword**
**Discuss it**
Question 3 Explanation:
i is assigned a value twice. Final variables can be assigned values only one. Following is the compiler error "Main.java:5: error: variable i might already have been assigned"
Question 4

```
class Base {
  public final void show() {
       System.out.println("Base::show() called");
    }
}
class Derived extends Base {
    public void show() {
       System.out.println("Derived::show() called");
    }
}
public class Main {
    public static void main(String[] args) {
        Base b = new Derived();;
        b.show();
    }
}
```

ADerived::show() called
BBase::show() called
CCompiler Error
DException
**final keyword**
**Discuss it**
Question 4 Explanation:
compiler error: show() in Derived cannot override show() in Base
There are 4 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

---

Question 1

Predict the output of following Java program

```
class Main {
   public static void main(String args[]) {
      try {
         throw 10;
      }
      catch(int e) {
         System.out.println("Got the  Exception " + e);
      }
  }
}
```

AGot the Exception 10
BGot the Exception 0
CCompiler Error
**Exception Handling**
**Discuss it**
Question 1 Explanation:
In Java only throwable objects (Throwable objects are instances of any subclass of the Throwable class) can be thrown as exception. So basic data type can no be thrown at all. Following are errors in the above program

```
Main.java:4: error: incompatible types
         throw 10;
               ^
  required: Throwable
  found:    int
Main.java:6: error: unexpected type
      catch(int e) {
            ^
  required: class
  found:    int
2 errors
```

Question 2

```
class Test extends Exception { }

class Main {
   public static void main(String args[]) {
      try {
         throw new Test();
      }
      catch(Test t) {
         System.out.println("Got the Test Exception");
      }
      finally {
         System.out.println("Inside finally block ");
```

```
        }
    }
}
```

A
```
Got the Test Exception
Inside finally block
```

B
```
Got the Test Exception
```

C
```
Inside finally block
```

DCompiler Error
**Exception Handling**
**Discuss it**
Question 2 Explanation:
In Java, the finally is always executed after the try-catch block. This block can be used to do the common cleanup work. There is no such block in C++.
Question 3
Output of following Java program?

```
class Main {
   public static void main(String args[]) {
      int x = 0;
      int y = 10;
      int z = y/x;
  }
}
```

ACompiler Error
BCompiles and runs fine
CCompiles fine but throws ArithmeticException exception
**Exception Handling**
**Discuss it**
Question 3 Explanation:
ArithmeticException is an unchecked exception, i.e., not checked by the compiler. So the program compiles fine. See following for more details. Checked vs Unchecked Exceptions in Java
Question 4

```
class Base extends Exception {}
class Derived extends Base  {}

public class Main {
  public static void main(String args[]) {
   // some other stuff
   try {
       // Some monitored code
       throw new Derived();
    }
    catch(Base b)     {
       System.out.println("Caught base class exception");
    }
    catch(Derived d)  {
       System.out.println("Caught derived class exception");
    }
  }
}
```

ACaught base class exception
BCaught derived class exception
CCompiler Error because derived is not throwable
DCompiler Error because base class exception is caught before derived class
**Exception Handling**
**Discuss it**
Question 4 Explanation:
See Catching base and derived classes as exceptions Following is the error in below program

```
Main.java:12: error: exception Derived has already been caught
    catch(Derived d)  { System.out.println("Caught derived class exception"); }
```

Question 5

```
class Test
{
    public static void main (String[] args)
    {
        try
        {
            int a = 0;
            System.out.println ("a = " + a + "\n");
            int b = 20 / a;
            System.out.println ("b = " + b);
        }

        catch(ArithmeticException e)
        {
            System.out.println ("Divide by zero error");
        }

        finally
        {
            System.out.println ("inside the finally block");
        }
    }
}
```

ACompile error
BDivide by zero error
C
```
a = 0
Divide by zero error
inside the finally block
```

Da = 0
Einside the finally block
**Exception Handling**
**Discuss it**
Question 5 Explanation:
On division of 20 by 0, divide by zero exception occurs and control goes inside the catch block. Also, the finally block is always executed whether an exception occurs or not.
Question 6

```
class Test
{
    public static void main(String[] args)
    {
        try
        {
            int a[]= {1, 2, 3, 4};
            for (int i = 1; i <= 4; i++)
            {
                System.out.println ("a[" + i + "]=" + a[i] + "\n");
            }
        }

        catch (Exception e)
        {
            System.out.println ("error = " + e);
        }

        catch (ArrayIndexOutOfBoundsException e)
        {
            System.out.println ("ArrayIndexOutOfBoundsException");
        }
    }
}
```

ACompiler error
BRun time error
CArrayIndexOutOfBoundsException
DError Code is printed
EArray is printed
**Exception Handling**
**Discuss it**
Question 6 Explanation:
ArrayIndexOutOfBoundsException has been already caught by base class Exception. When a subclass exception is mentioned after base class exception, then error occurs.
Question 7
Predict the output of the following program.

```
class Test
{
    String str = "a";

    void A()
    {
        try
        {
            str +="b";
            B();
        }
        catch (Exception e)
        {
            str += "c";
        }
    }

    void B() throws Exception
    {
        try
        {
            str += "d";
            C();
        }
        catch(Exception e)
        {
            throw new Exception();
        }
        finally
        {
            str += "e";
        }

        str += "f";

    }

    void C() throws Exception
    {
        throw new Exception();
    }

    void display()
    {
        System.out.println(str);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.A();
        object.display();
    }

}
```

Aabdef
Babdec
Cabdefc
**Exception Handling**
**Discuss it**
Question 7 Explanation:
'throw' keyword is used to explicitly throw an exception. finally block is always executed even when an exception occurs. Call to method C() throws an exception. Thus, control goes in catch block of method B() which again throws an exception. So, control goes in catch block of method A().
Question 8
Predict the output of the following program.

```
class Test
{   int count = 0;

    void A() throws Exception
    {
        try
        {
            count++;

            try
            {
                count++;

                try
                {
                    count++;
                    throw new Exception();

                }

                catch(Exception ex)
                {
                    count++;
                    throw new Exception();
                }
            }

            catch(Exception ex)
            {
                count++;
            }
        }

        catch(Exception ex)
        {
            count++;
        }

    }

    void display()
    {
        System.out.println(count);
    }

    public static void main(String[] args) throws Exception
    {
        Test obj = new Test();
        obj.A();
        obj.display();
    }
}
```

A4
B5
C6
DCompilation error
**Exception Handling**
**Discuss it**
Question 8 Explanation:
'throw' keyword is used to explicitly throw an exception. In third try block, exception is thrown. So, control goes in catch block. Again, in catch block exception is thrown. So, control goes in inner catch block.
There are 8 questions to complete.

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.

---

Question 1
Which of the following is/are true about packages in Java?

```
1) Every class is part of some package.
2) All classes in a file are part of the same package.
3) If no package is specified, the classes in the file
   go into a special unnamed package
4) If no package is specified, a new package is created with
   folder name of class and the class is put in this package.
```

AOnly 1, 2 and 3
BOnly 1, 2 and 4
COnly 4
DOnly 1 and 3
**Java Packages**
**Discuss it**
Question 1 Explanation:
In Java, a package can be considered as equivalent to C++ language's namespace.
Question 2
Which of the following is/are advantages of packages?
APackages avoid name clashes
BClasses, even though they are visible outside their package, can have fields visible to packages only
CWe can have hidden classes that are used by the packages, but not visible outside.
DAll of the above
**Java Packages**
**Discuss it**
Question 3
Predict the output of following Java program

```
// Note static keyword after import.
import static java.lang.System.*;

class StaticImportDemo
{
   public static void main(String args[])
   {
       out.println("GeeksforGeeks");
   }
}
```

ACompiler Error
BRuntime Error
CGeeksforGeeks
DNone of the above
**Java Packages**
**Discuss it**
Question 3 Explanation:
Please refer http://www.geeksforgeeks.org/packages-in-java/
There are 3 questions to complete.

Please do Like/Share if you find the above useful. Also, please do leave us comment for further clarification or info. We would love to help and learn

## GATE CS Corner

See Placement Course for placement preparation, GATE Corner for GATE CS Preparation and Quiz Corner for all Quizzes on GeeksQuiz.