## **What is Java**

Java is a **programming language** and a **platform**.

Java is a high level, robust, object-oriented and secure programming language.

**Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

## **Application**

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

## **Types of Java Applications**

There are mainly 4 types of applications that can be created using Java programming:

#### **1) Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

#### **2) Web Application**

An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. technologies are used for creating web applications in Java.

#### **3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

#### **4) Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

## **Java Platforms / Editions**

There are 4 platforms or editions of Java:

#### **1) Java SE (Java Standard Edition)**

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, [String](https://www.javatpoint.com/java-string), Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

#### **2) Java EE (Java Enterprise Edition)**

It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, [JPA](https://www.javatpoint.com/jpa-tutorial), etc.

#### **3) Java ME (Java Micro Edition)**

It is a micro platform which is mainly used to develop mobile applications.

#### **4) JavaFX**

It is used to develop rich internet applications. It uses a light-weight user interface API.

**JDK** – Java Development Kit (in short JDK) is Kit which provides the environment to develop and execute(run) the Java program. JDK is a kit (or package) which includes two things

* + 1. Development Tools (to provide an environment to develop your java programs)
    2. JRE (to execute your java program).

Note  JDK is only used by Java Developers.

* JRE – Java Runtime Environment (to say JRE) is an installation package which provides environment to only run ( not develop) the java program(or application)onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
* JVM – Java Virtual machine(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line hence it is also known as interpreter.

**How does JRE and JDK works?**

**What does JRE consists of?**  
JRE consists of the following components:

* Deployment technologies, including deployment, Java Web Start and Java Plug-in.
* User interface toolkits, including *Abstract Window Toolkit (AWT), Swing, Java 2D, Accessibility, Image I/O, Print Service, Sound, drag and drop (DnD)* and *input methods*.
* Integration libraries, including *Interface Definition Language (IDL), Java Database Connectivity (JDBC), Java Naming and Directory Interface (JNDI), Remote Method Invocation (RMI), Remote Method Invocation Over Internet Inter-Orb Protocol (RMI-IIOP)* and *scripting*.
* Other base libraries, including *international support, input/output (I/O), extension mechanism, Beans, Java Management Extensions (JMX), Java Native Interface (JNI), Math, Networking, Override Mechanism, Security, Serialization* and *Java for XML Processing (XML JAXP)*.
* Lang and util base libraries, including *lang and util, management, versioning, zip, instrument, reflection, Collections, Concurrency Utilities, Java Archive (JAR), Logging, Preferences API, Ref Objects* and *Regular Expressions*.
* **Java Virtual Machine (JVM)**,

including *Java HotSpot Client* and *Server Virtual Machines*.

**How does JRE works?**  
To understand how the JRE works let us consider a Java source file saved as *Example.java*. The file is compiled into a set of Byte Code that is stored in a “*.class*” file. Here it will be “*Example.class*“.  
  
The following diagram depicts what is done at compile

time.

The following actions occur at runtime.

* **Class Loader**

The Class Loader loads all necessary classes needed for the execution of a program. It provides security by separating the namespaces of the local file system from that imported through the network. These files are loaded either from a hard disk, a network or from other sources.

* **Byte Code Verifier**

The JVM puts the code through the Byte Code Verifier that checks the format and checks for an illegal code. Illegal code, for example, is code that violates access rights on objects or violates the implementation of pointers.

The Byte Code verifier ensures that the code adheres to the JVM specification and does not violate system integrity.

* **Intrepreter**

At runtime the Byte Code is loaded, checked and run by the interpreter. The interpreter has the following two functions:

* + Execute the Byte Code
  + Make appropriate calls to the underlying hardware



Both operations can be shown as:  
  
To understand the interactions between JDK and JRE consider the following diagram.

**How does JVM works?**

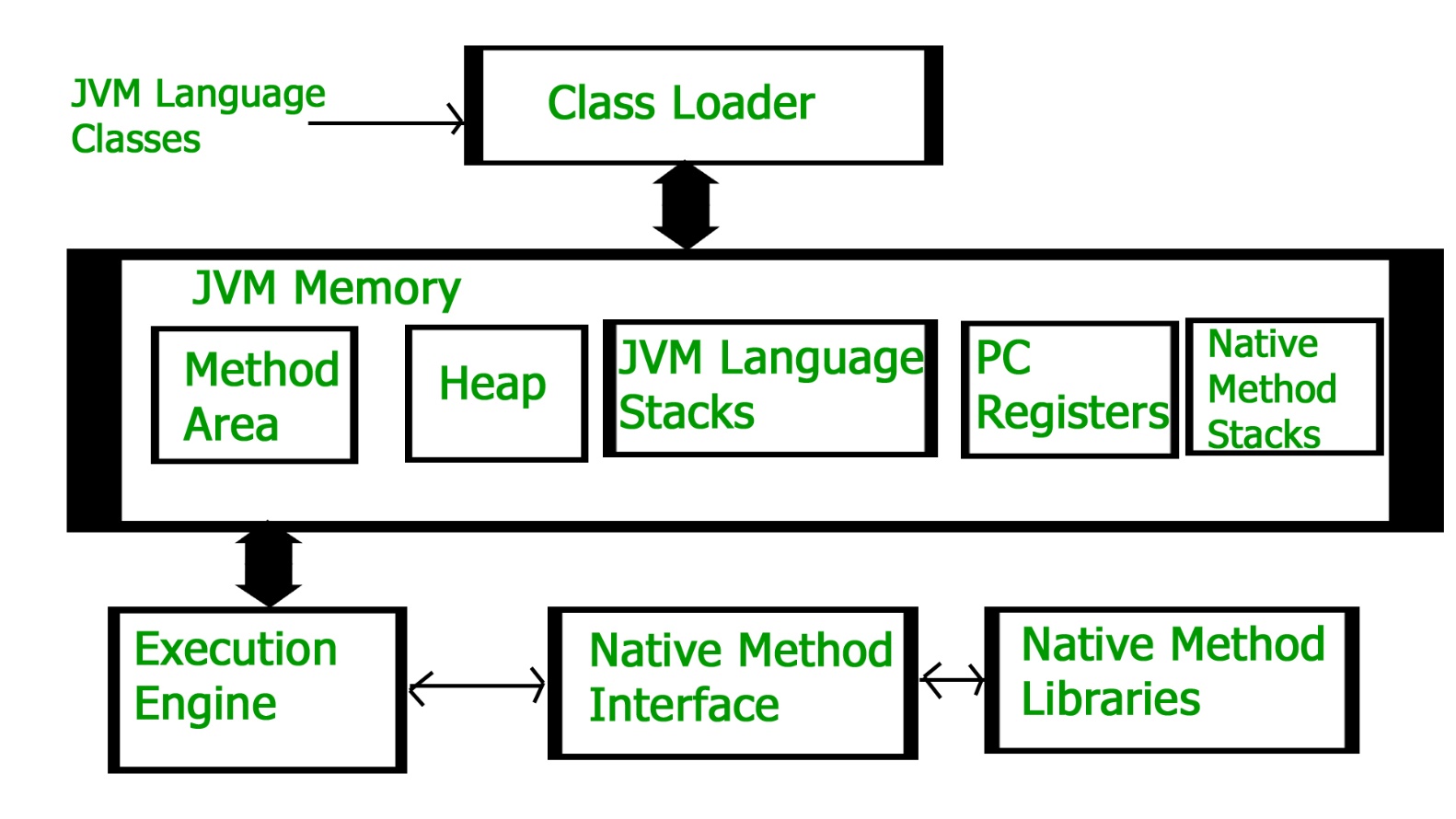
JVM becomes an instance of JRE at runtime of a Java program. It is widely known as a runtime interpreter. JVM largely helps in the abstraction of inner implementation from the programmers who make use of libraries for their programmes from JDK.

**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 Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). 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.

**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](https://www.geeksforgeeks.org/object-class-in-java/) 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  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;      }  } |

Run on IDE

Output:

Student

getName setName getRoll\_no setRoll\_no name roll\_No

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

**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 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$ExtClassLoader* class.
* *System/Application class loader* : It is child of extension class loader. It is responsible to load classes from application class path. It internally uses Environment Variable which mapped to java.class.path. It is also implemented in Java by the *sun.misc.Launcher$AppClassLoader*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());      }  } |

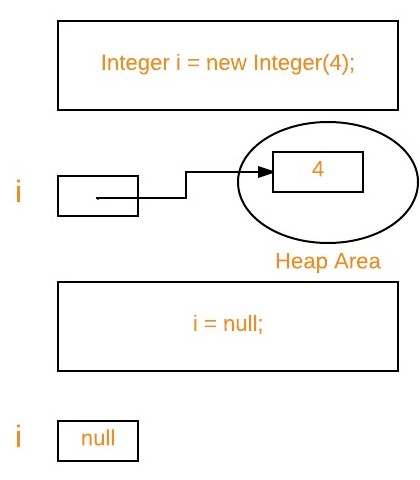
Garbage Collection in Java

**Introduction**

* In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **Out of Memory Errors**.
* But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
* Garbage collector is best example of [Daemon thread](https://www.geeksforgeeks.org/daemon-thread-java/) as it is always running in background.
* Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

**Important terms:**

1. **Unreachable objects:**An object is said to be unreachable if it doesn’t contain any reference to it. Also note that objects which are part of [island of isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/) are also unreachable.
2. Integer i = new Integer(4);
3. // the new Integer object is reachable via the reference in 'i'
4. i = null;
5. // the Integer object is no longer reachable.



1. **Eligibility for garbage collection :**An object is said to be eligible for GC(garbage collection) if it is unreachable.
2. In above image, after *i = null;* integer object 4 in heap area is eligible for garbage collection.

**Ways to make an object eligible for GC**

* Even though programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
* There are generally four different ways to make an object eligible for garbage collection.
  + 1. Nullifying the reference variable
    2. Re-assigning the reference variable
    3. Object created inside method
    4. [Island of Isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/)

**Ways for requesting**[**JVM**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)**to run Garbage Collector**

* Once we made object eligible for garbage collection, it may not destroy immediately by garbage collector. Whenever JVM runs Garbage Collector program, then only object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
* We can also request JVM to run Garbage Collector. There are two ways to do it :
  + 1. **Using *System.gc()* method** : System class contain static method *gc()* for requesting JVM to run Garbage Collector.
    2. **Using *Runtime.getRuntime().gc()* method** : [Runtime class](https://www.geeksforgeeks.org/java-lang-runtime-class-in-java/) allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

|  |
| --- |
| // Java program to demonstrate requesting  // JVM to run Garbage Collector  public class Test  {      public static void main(String[] args) throws InterruptedException      {          Test t1 = new Test();          Test t2 = new Test();            // Nullifying the reference variable          t1 = null;            // requesting JVM for running Garbage Collector          System.gc();            // Nullifying the reference variable          t2 = null;            // requesting JVM for running Garbage Collector          Runtime.getRuntime().gc();        }        @Override      // finalize method is called on object once      // before garbage collecting it      protected void finalize() throws Throwable      {          System.out.println("Garbage collector called");          System.out.println("Object garbage collected : " + this);      }  } |

* **Garbage collector called**
* **Object garbage collected : Test@46d08f12**
* **Garbage collector called**
* **Object garbage collected : Test@481779b8**
* **Note :**
  + 1. **There is no guarantee that any one of above two methods will definitely run Garbage Collector.**
    2. **The call *System.gc()* is effectively equivalent to the call : *Runtime.getRuntime().gc()***

# sIsland of Isolation in Java

In java, object destruction is taken care by the [Garbage Collector](https://www.geeksforgeeks.org/garbage-collection-java/) 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");      }  } |

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.

## **finalize() method**

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected** **void** finalize(){}

#### **Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).**

## **gc() method**

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public** **static** **void** gc(){}

#### **Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before object is garbage collected.**

### **Simple Example of garbage collection in java**

1. **public** **class** TestGarbage1{
2. **public** **void** finalize(){System.out.println("object is garbage collected");}
3. **public** **static** **void** main(String args[]){
4. TestGarbage1 s1=**new** TestGarbage1();
5. TestGarbage1 s2=**new** TestGarbage1();
6. s1=**null**;
7. s2=**null**;
8. System.gc();
9. }
10. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestGarbage1)

object is garbage collected

object is garbage collected

#### **Note: Neither finalization nor garbage collection is guaranteed.**

# Java String In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);
3. String s="javatpoint";

**Java String** class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements Serializable, Comparable and CharSequence interfaces.

## **CharSequence Interface**

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.

The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

We will discuss immutable string later. Let's first understand what is String in Java and how to create the String object.

There are two ways to create String object:

1. By string literal
2. By new keyword

Each time you create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance

### **2) By new keyword**

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

Difference between method overloading and method overriding in java

|  |  |  |
| --- | --- | --- |
| **No.** | **Method Overloading** | **Method Overriding** |
| 1) | Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| 2) | Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| 3) | In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| 4) | Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| 5) | In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | *Return type must be same or covariant* in method overriding. |

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | [char charAt(int index)](https://www.javatpoint.com/java-string-charat) | returns char value for the particular index |
| 2 | [int length()](https://www.javatpoint.com/java-string-length) | returns string length |
| 3 | [static String format(String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns a formatted string. |
| 4 | [static String format(Locale l, String format, Object... args)](https://www.javatpoint.com/java-string-format) | returns formatted string with given locale. |
| 5 | [String substring(int beginIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index. |
| 6 | [String substring(int beginIndex, int endIndex)](https://www.javatpoint.com/java-string-substring) | returns substring for given begin index and end index. |
| 7 | [boolean contains(CharSequence s)](https://www.javatpoint.com/java-string-contains) | returns true or false after matching the sequence of char value. |
| 8 | [static String join(CharSequence delimiter, CharSequence... elements)](https://www.javatpoint.com/java-string-join) | returns a joined string. |
| 9 | [static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)](https://www.javatpoint.com/java-string-join) | returns a joined string. |
| 10 | [boolean equals(Object another)](https://www.javatpoint.com/java-string-equals) | checks the equality of string with the given object. |
| 11 | [boolean isEmpty()](https://www.javatpoint.com/java-string-isempty) | checks if string is empty. |
| 12 | [String concat(String str)](https://www.javatpoint.com/java-string-concat) | concatenates the specified string. |
| 13 | [String replace(char old, char new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of the specified char value. |
| 14 | [String replace(CharSequence old, CharSequence new)](https://www.javatpoint.com/java-string-replace) | replaces all occurrences of the specified CharSequence. |
| 15 | [static String equalsIgnoreCase(String another)](https://www.javatpoint.com/java-string-equalsignorecase) | compares another string. It doesn't check case. |
| 16 | [String[] split(String regex)](https://www.javatpoint.com/java-string-split) | returns a split string matching regex. |
| 17 | [String[] split(String regex, int limit)](https://www.javatpoint.com/java-string-split) | returns a split string matching regex and limit. |
| 18 | [String intern()](https://www.javatpoint.com/java-string-intern) | returns an interned string. |
| 19 | [int indexOf(int ch)](https://www.javatpoint.com/java-string-indexof) | returns the specified char value index. |
| 20 | [int indexOf(int ch, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns the specified char value index starting with given index. |
| 21 | [int indexOf(String substring)](https://www.javatpoint.com/java-string-indexof) | returns the specified substring index. |
| 22 | [int indexOf(String substring, int fromIndex)](https://www.javatpoint.com/java-string-indexof) | returns the specified substring index starting with given index. |
| 23 | [String toLowerCase()](https://www.javatpoint.com/java-string-tolowercase) | returns a string in lowercase. |
| 24 | [String toLowerCase(Locale l)](https://www.javatpoint.com/java-string-tolowercase) | returns a string in lowercase using specified locale. |
| 25 | [String toUpperCase()](https://www.javatpoint.com/java-string-touppercase) | returns a string in uppercase. |
| 26 | [String toUpperCase(Locale l)](https://www.javatpoint.com/java-string-touppercase) | returns a string in uppercase using specified locale. |
| 27 | [String trim()](https://www.javatpoint.com/java-string-trim) | removes beginning and ending spaces of this string. |
| 28 | [static String valueOf(int value)](https://www.javatpoint.com/java-string-valueof) | converts given type into string. It is an overloaded method. |

### **Java String class methods**

# Java StringBuffer class

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringBuffer() | creates an empty string buffer with the initial capacity of 16. |
| StringBuffer(String str) | creates a string buffer with the specified string. |
| StringBuffer(int capacity) | creates an empty string buffer with the specified capacity as length. |

### **Important methods of StringBuffer class**

|  |  |  |
| --- | --- | --- |
| **Modifier and Type** | **Method** | **Description** |
| public synchronized StringBuffer | append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public synchronized StringBuffer | insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public synchronized StringBuffer | replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public synchronized StringBuffer | reverse() | is used to reverse the string. |
| public int | capacity() | is used to return the current capacity. |
| public void | ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char | charAt(int index) | is used to return the character at the specified position. |
| public int | length() | is used to return the length of the string i.e. total number of characters. |
| public String | substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String | substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

1. **class** StringBuffer**Example**{
2. **public** **static** **void** main(String args[]){
3. StringBuffer sb=**new** StringBuffer("Hello ");
4. sb.append("Java");//now original string is changed
5. System.out.println(sb);//prints Hello Java
6. }
7. }
8. **class** **StringBufferExample2{**
9. **public** **static** **void** main(String args[]){
10. StringBuffer sb=**new** StringBuffer("Hello ");
11. sb.insert(1,"Java");//now original string is changed
12. System.out.println(sb);//prints HJavaello
13. }
14. }
15. **class** S**tringBufferExample3**{
16. **public** **static** **void** main(String args[]){
17. StringBuffer sb=**new** StringBuffer("Hello");
18. sb.replace(1,3,"Java");
19. System.out.println(sb);//prints HJavalo
20. }
21. }
22. class StringBufferExample4{
23. **public** **static** **void** main(String args[]){
24. StringBuffer sb=**new** StringBuffer("Hello");
25. sb.delete(1,3);
26. System.out.println(sb);//prints Hlo
27. }
28. }
29. class StringBufferExample5{
30. **public** **static** **void** main(String args[]){
31. StringBuffer sb=**new** StringBuffer("Hello");
32. sb.reverse();
33. System.out.println(sb);//prints olleH
34. }
35. }
36. **class** **StringBufferExample6**{
37. **public** **static** **void** main(String args[]){
38. StringBuffer sb=**new** StringBuffer();
39. System.out.println(sb.capacity());//default 16
40. sb.append("Hello");
41. System.out.println(sb.capacity());//now 16
42. sb.append("java is my favourite language");
43. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
44. }
45. **class** StringBufferExample7{
46. **public** **static** **void** main(String args[]){
47. StringBuffer sb=**new** StringBuffer();
48. System.out.println(sb.capacity());//default 16
49. sb.append("Hello");
50. System.out.println(sb.capacity());//now 16
51. sb.append("java is my favourite language");
52. System.out.println(sb.capacity());//now (16\*2)+2=34 i.e (oldcapacity\*2)+2
53. sb.ensureCapacity(10);//now no change
54. System.out.println(sb.capacity());//now 34
55. sb.ensureCapacity(50);//now (34\*2)+2
56. System.out.println(sb.capacity());//now 70
57. }
58. }

## **Important methods of StringBuilder class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc. |
| public StringBuilder insert(int offset, String s) | is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc. |
| public StringBuilder replace(int startIndex, int endIndex, String str) | is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | is used to reverse the string. |
| public int capacity() | is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | is used to return the character at the specified position. |
| public int length() | is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | is used to return the substring from the specified beginIndex and endIndex. |

# Difference between String and StringBuffer

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | String class is immutable. | StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when you concat too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when you concat strings. |
| 3) | String class overrides the equals() method of Object class. So you can compare the contents of two strings by equals() method. | StringBuffer class doesn't override the equals() method of Object class. |

|  |  |  |
| --- | --- | --- |
| **No.** | **StringBuffer** | **StringBuilder** |
| 1) | StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously. | StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously. |
| 2) | StringBuffer is *less efficient* than StringBuilder. | StringBuilder is *more efficient* than StringBuffer. |

# StringTokenizer in Java

1. [StringTokenizer](https://www.javatpoint.com/string-tokenizer-in-java)
2. [Methods of StringTokenizer](https://www.javatpoint.com/string-tokenizer-in-java)
3. [Example of StringTokenizer](https://www.javatpoint.com/string-tokenizer-in-java)

The **java.util.StringTokenizer** class allows you to break a string into tokens. It is simple way to break string.

It doesn't provide the facility to differentiate numbers, quoted strings, identifiers etc. like StreamTokenizer class.

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| StringTokenizer(String str) | creates StringTokenizer with specified string. |
| StringTokenizer(String str, String delim) | creates StringTokenizer with specified string and delimeter. |
| StringTokenizer(String str, String delim, boolean returnValue) | creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens. |

#### **Methods of StringTokenizer class**

The 6 useful methods of StringTokenizer class are as follows:

|  |  |
| --- | --- |
| **Public method** | **Description** |
| boolean hasMoreTokens() | checks if there is more tokens available. |
| String nextToken() | returns the next token from the StringTokenizer object. |
| String nextToken(String delim) | returns the next token based on the delimeter. |
| boolean hasMoreElements() | same as hasMoreTokens() method. |
| Object nextElement() | same as nextToken() but its return type is Object. |
| int countTokens() | returns the total number of tokens. |

1. **import** java.util.StringTokenizer;
2. **public** **class** Simple{
3. **public** **static** **void** main(String args[]){
4. StringTokenizer st = **new** StringTokenizer("my name is khan"," ");
5. **while** (st.hasMoreTokens()) {
6. System.out.println(st.nextToken());
7. }
8. }

}

1. Output:my
2. name
3. is
4. khan
5. **ublic** **class** Test {
6. **public** **static** **void** main(String[] args) {
7. StringTokenizer st = **new** StringTokenizer("my,name,is,khan");
9. // printing next token
10. System.out.println("Next token is : " + st.nextToken(","));
11. }
12. }
13. Output:Next token is : my

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## **Usage of Java super Keyword**

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

## **3) super is used to invoke parent class constructor.**

1. **class** Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. **class** Dog **extends** Animal{
5. Dog(){
6. **super**();
7. System.out.println("dog is created");
8. }
9. }
10. **class** TestSuper3{
11. **public** **static** **void** main(String args[]){
12. Dog d=**new** Dog();
13. }}
14. **Note: super() is added in each class constructor automatically by compiler if there is no super() or this().**

If above example super () is skip still we get same output as above program.

### **Abstraction in Java**

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### **Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%) 2 Interface (100%)

**Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### **Points to Remember**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.
* **abstract** **class** Bike{
* **abstract** **void** run();
* }
* **class** Honda4 **extends** Bike{
* **void** run(){System.out.println("running safely");}
* **public** **static** **void** main(String args[]){
* Bike obj = **new** Honda4();
* obj.run();
* }
* }

**Example**

1. **abstract** **class** Bank{
2. **abstract** **int** getRateOfInterest();
3. }
4. **class** SBI **extends** Bank{
5. **int** getRateOfInterest(){**return** 7;}
6. }
7. **class** PNB **extends** Bank{
8. **int** getRateOfInterest(){**return** 8;}
9. }
11. **class** TestBank{
12. **public** **static** **void** main(String args[]){
13. Bank b;
14. b=**new** SBI();
15. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
16. b=**new** PNB();
17. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
18. }}

#### **If there is an abstract method in a class, that class must be abstract.**

#### **If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.**

1. **interface** A{
2. **void** a();  **void** b(); **void** c();  **void** d();
3. }
5. **abstract** **class** B **implements** A{
6. **public** **void** c(){System.out.println("I am c");}
7. }
9. **class** M **extends** B{
10. **public** **void** a(){System.out.println("I am a");}
11. **public** **void** b(){System.out.println("I am b");}
12. **public** **void** d(){System.out.println("I am d");}
13. }
15. **class** Test5{
16. **public** **static** **void** main(String args[]){
17. A a=**new** M();
18. a.a();  a.b();  a.c();  a.d();
19. }}

# Interface in Java

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Java Interface also **represents the IS-A relationship**.

It cannot be instantiated just like the abstract class.

Since Java 8, we can have **default and static methods** in an interface.

Since Java 9, we can have **private methods** in an interface.

There are mainly three reasons to **use** interface. They are given below.

* It is used to achieve abstraction.
* By interface, we can support the functionality of multiple inheritance.
* It can be used to achieve loose coupling.

#### **The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.**

## **Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?**

As we have explained in the inheritance chapter, multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class. For example:

1. **interface** Printable{
2. **void** print();
3. }
4. **interface** Showable{
5. **void** print();
6. }
7. **class** TestInterface3 **implements** Printable, Showable{
8. **public** **void** print(){System.out.println("Hello");}
9. **public** **static** **void** main(String args[]){
10. TestInterface3 obj = **new** TestInterface3();
11. obj.print();
12. }
13. }

## **Java 8 Default Method in Interface**

Since Java 8, we can have method body in interface. But we need to make it default method. Let's see an example:

1. **interface** Drawable{
2. **void** draw();
3. **default** **void** msg(){System.out.println("default method");}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** TestInterfaceDefault{
9. **public** **static** **void** main(String args[]){
10. Drawable d=**new** Rectangle();
11. d.draw();
12. d.msg();
13. }}

Output: drawing rectangle default method

|  |  |
| --- | --- |
| **Abstract class** | **Interface** |
| 1) Abstract class can **have abstract and non-abstract** methods. | Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also. |
| 2) Abstract class **doesn't support multiple inheritance**. | Interface **supports multiple inheritance**. |
| 3) Abstract class **can have final, non-final, static and non-static variables**. | Interface has **only static and final variables**. |
| 4) Abstract class **can provide the implementation of interface**. | Interface **can't provide the implementation of abstract class**. |
| 5) The **abstract keyword** is used to declare abstract class. | The **interface keyword** is used to declare interface. |
| 6) An **abstract class** can extend another Java class and implement multiple Java interfaces. | An **interface** can extend another Java interface only. |
| 7) An **abstract class** can be extended using keyword "extends". | An **interface class** can be implemented using keyword "implements". |
| 8) A Java **abstract class** can have class members like private, protected, etc. | Members of a Java interface are public by default. |
| 9)**Example:** public abstract class Shape{ public abstract void draw(); } | **Example:** public interface Drawable{ void draw(); } |

## **Java 8 Static Method in Interface**

1. **interface** Drawable{
2. **void** draw();
3. **static** **int** cube(**int** x){**return** x\*x\*x;}
4. }
5. **class** Rectangle **implements** Drawable{
6. **public** **void** draw(){System.out.println("drawing rectangle");}
7. }
8. **class** TestInterfaceStatic{
9. **public** **static** **void** main(String args[]){
10. Drawable d=**new** Rectangle();
11. d.draw();
12. System.out.println(**Drawable.cube(3)**);
13. }}

Output: drawing rectangle 27

An interface which has no member is known as **a marker or tagged interface,** for example, Serializable, Clone able, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

The class (method) area stores *code* - that's the code of your program. The heap stores object instances. For example:

public void MakeSomeFruit(){

Fruit myFruit=new Fruit();

}

* The MakeSomeFruit code is stored in the class area.
* When executed, the actual Fruit instance it creates is stored in the Heap.
* When executed, the myFruit *reference* variable is stored on the stack. That's just a number which points at the location of the instance in memory - an address.

Inside a Java virtual machine instance, information about loaded types is stored in a logical area of memory called the method area. When the Java virtual machine loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file--a linear stream of binary data--and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (static) variables declared in the class is also taken from the method area.

Java 7 (codename “Dolphin”)

**Major features of Java 7 include**:

* Language enhancements grouped under a Project Coin
* String object in switch statement
* Multiple exceptions handling to eliminate duplication of codes
* Upgraded class-loader architecture
* Improved type interference for generic instance
* Library support for ECC (elliptic curve cryptography) algorithms
* Upgraded Rowset 1.1 and JDBC 4.1
* Improved Managed Beans
* Automatic resource management in try-statement
* Concurrency and collections updates
* Compressed 64-bit pointers
* JVM support for dynamically-typed languages

**Major features of Java 8 include:**

* Language-level support for Lambda Expressions
* Interface default and Static Methods
* Unsigned Integer Arithmetic
* Concurrent API enhancements
* New Date and Time API
* Parallel Sorting
* Null Reference Template
* New JavaScript Engine, Nashorn
* New and improved Stream API
* Removal of permanent generation

### Difference between Java 7 and Java 8

1. Java SE 7 was the first and the major update to the programming language under the ownership and stewardship of Oracle Corporation after it acquired Sun Microsystems in 2010. Java 8, on the other hand, was the next biggest update after Java 7 which accommodates some major features and enhancements to the Java model.
2. One of the most notable features added in the Java 7 is the [JVM](http://www.differencebetween.net/technology/software-technology/difference-between-jvm-and-jre/) support for dynamically-typed languages plus small language enhancements (Project Coin). Java 8 is a major update to the programming language which introduced a significant upgrade to the functional programming called the Lambda Expressions.
3. Java 8 also gets a new and improved Date/Time API, an enhanced JavaScript engine, new streaming API. Concurrent accumulators, secure random generation, and much more. Java 7 had improved class-loader architecture, enhanced Managed Beans, multiple exceptions handling support, etc.
4. Java 7 added a language support for String in switch so users can rewrite codes more efficiently and easily. This helps users write better readable codes. The Lambda Expressions in Java 8 allow you to code local functions as method arguments.
5. Oracle stopped dispersing security updates for Java SE 7 effective April 2015, however existing downloads as remain accessible from the Java Archive. Java 8 public downloads will reportedly be stopped after Sept 2018.

### Java 7 vs. Java 8: Comparison Chart to show the difference between Java 7 and Java 8

|  |  |
| --- | --- |
| **Java 7** | **Java 8** |
| Java SE 7 was codenamed Dolphin. | Code name for Java SE 8 is Spider. |
| Java 7 is supported on Win XP. | Java 8 is not officially supported on Win XP. |
| Java 7 brings JVM support for dynamically-typed languages plus Type Interference for Generic Instance creation. | Java 8 brings the most anticipated feature for the programming language called Lambda Expressions, a new language feature which allows users to code local functions as method arguments. |
| Users can catch multiple exception types in one catch block which could be impossible before JDK 7. | Java 8 brings its own new specialized API for Date and Time manipulation. |
| Small language enhancements was brought to simplify common programming tasks such as automatic resource management, string object in switch, better exception handling, etc. | New and improved JavaScript engine, Nashorn which allows developers to run the script on a JVM. The idea was to implement a lightweight JavaScript runtime in the programming language with a native JVM. |

### Read more: [Difference between Java 7 and Java 8 | Difference Between](http://www.differencebetween.net/technology/difference-between-java-7-and-java-8/#ixzz5S66wZL4k) <http://www.differencebetween.net/technology/difference-between-java-7-and-java-8/#ixzz5S66wZL4k> http://docs.oracle.com/javase/7/docs/api/. **Understanding meaningful example of Aggregation**

In this example, Employee has an object of Address, address object contains its own informations such as city, state, country etc. In such case relationship is Employee HAS-A address.

#### **Address.java**

1. **public** **class** Address {
2. String city,state,country;
4. **public** Address(String city, String state, String country) {
5. **this**.city = city;
6. **this**.state = state;
7. **this**.country = country;
8. }
10. }

#### **Emp.java**

1. **public** **class** Emp {
2. **int** id;
3. String name;
4. Address address;
6. **public** Emp(**int** id, String name,Address address) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.address=address;
10. }
12. **void** display(){
13. System.out.println(id+" "+name);
14. System.out.println(address.city+" "+address.state+" "+address.country);
15. }
17. **public** **static** **void** main(String[] args) {
18. Address address1=**new** Address("gzb","UP","india");
19. Address address2=**new** Address("gno","UP","india");
21. Emp e=**new** Emp(111,"varun",address1);
22. Emp e2=**new** Emp(112,"arun",address2);
24. e.display();
25. e2.display();
27. }
28. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Emp)

Output:111 varun

gzb UP india

112 arun

gno UP india

### **Advantages of Java Multithreading**

1) It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

### **1) Process-based Multitasking (Multiprocessing)**

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

### **2) Thread-based Multitasking (Multithreading)**

* Threads share the same address space.
* A thread is lightweight.
* Cost of communication between the thread is low.

#### **Note: At least one process is required for each thread.**

A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.

But for better understanding the threads, we are explaining it in the 5 states.

The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:

1. New
2. Runnable
3. Running
4. Non-Runnable (Blocked)
5. Terminated

There are two ways to create a thread:

1. By extending Thread class
2. By implementing Runnable interface.

### **Thread class:**

|  |
| --- |
| Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface. |

### **Commonly used Constructors of Thread class:**

|  |
| --- |
| * Thread() * Thread(String name) * Thread(Runnable r) * Thread(Runnable r,String name) |

### **Commonly used methods of Thread class:**

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. 2. **public void start():**starts the execution of the thread.JVM calls the run() method on the thread. 3. **public void sleep(long miliseconds):**Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds. 4. **public void join():**waits for a thread to die. 5. **public void join(long miliseconds):**waits for a thread to die for the specified miliseconds. 6. **public int getPriority():**returns the priority of the thread. 7. **public int setPriority(int priority):**changes the priority of the thread. 8. **public String getName():**returns the name of the thread. 9. **public void setName(String name):**changes the name of the thread. 10. **public Thread currentThread():**returns the reference of currently executing thread. 11. **public int getId():**returns the id of the thread. 12. **public Thread.State getState():**returns the state of the thread. 13. **public boolean isAlive():**tests if the thread is alive. 14. **public void yield():**causes the currently executing thread object to temporarily pause and allow other threads to execute. 15. **public void suspend():**is used to suspend the thread(depricated). 16. **public void resume():**is used to resume the suspended thread(depricated). 17. **public void stop():**is used to stop the thread(depricated). 18. **public boolean isDaemon():**tests if the thread is a daemon thread. 19. **public void setDaemon(boolean b):**marks the thread as daemon or user thread. 20. **public void interrupt():**interrupts the thread. 21. **public boolean isInterrupted():**tests if the thread has been interrupted. 22. **public static boolean interrupted():**tests if the current thread has been interrupted. |

### **Runnable interface:**

|  |
| --- |
| The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). |

|  |
| --- |
| 1. **public void run():**is used to perform action for a thread. |

### **Starting a thread:**

|  |
| --- |
| **start() method** of Thread class is used to start a newly created thread. It performs following tasks:   * A new thread starts(with new callstack). * The thread moves from New state to the Runnable state. * When the thread gets a chance to execute, its target run() method will run. |

### **1) Java Thread Example by extending Thread class**

1. **class** Multi **extends** Thread{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
5. **public** **static** **void** main(String args[]){
6. Multi t1=**new** Multi();
7. t1.start();
8. }
9. }

Output:thread is running...

### **2) Java Thread Example by implementing Runnable interface**

1. **class** Multi3 **implements** Runnable{
2. **public** **void** run(){
3. System.out.println("thread is running...");
4. }
6. **public** **static** **void** main(String args[]){
7. Multi3 m1=**new** Multi3();
8. Thread t1 =**new** Thread(m1);
9. t1.start();
10. }
11. }

Output:thread is running...

|  |
| --- |
| If you are not extending the Thread class,your class object would not be treated as a thread object.So you need to explicitely create Thread class object.We are passing the object of your class that implements Runnable so that your class run() method may execute |

# Thread Scheduler in Java

**Thread scheduler** in java is the part of the JVM that decides which thread should run.

There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.

Only one thread at a time can run in a single process.

The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

### **Difference between preemptive scheduling and time slicing**

Under preemptive scheduling, the highest priority task executes until it enters the waiting or dead states or a higher priority task comes into existence. Under time slicing, a task executes for a predefined slice of time and then reenters the pool of ready tasks. The scheduler then determines which task should execute next, based on priority and other factors.

**What happens if we start a thread twice?**

No. After starting a thread, it can never be started again. If you does so, an *IllegalThreadStateException* is thrown. In such case, thread will run once but for second time, it will throw exception.

Let's understand it by the example given below:

1. **public** **class** TestThreadTwice1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestThreadTwice1 t1=**new** TestThreadTwice1();
7. t1.start();
8. t1.start();
9. }
10. }

What if we call run() method directly instead start() method?

|  |
| --- |
| * Each thread starts in a separate call stack. * Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack |

1. **class** TestCallRun2 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<5;i++){
4. **try**{Thread.sleep(500);}**catch**(InterruptedException e){System.out.println(e);}
5. System.out.println(i);
6. }
7. }
8. **public** **static** **void** main(String args[]){
9. TestCallRun2 t1=**new** TestCallRun2();
10. TestCallRun2 t2=**new** TestCallRun2();
12. t1.run();
13. t2.run();
14. }
15. }

# The join() method

The join() method waits for a thread to die. In other words, it causes the currently running threads to stop executing until the thread it joins with completes its task.

1. **class** TestJoinMethod1 **extends** Thread{
2. **public** **void** run(){
3. **for**(**int** i=1;i<=5;i++){
4. **try**{
5. Thread.sleep(500);
6. }**catch**(Exception e){System.out.println(e);}
7. System.out.println(i);
8. }
9. }
10. **public** **static** **void** main(String args[]){
11. TestJoinMethod1 t1=**new** TestJoinMethod1();
12. TestJoinMethod1 t2=**new** TestJoinMethod1();
13. TestJoinMethod1 t3=**new** TestJoinMethod1();
14. t1.start();
15. **try**{
16. t1.join();
17. }**catch**(Exception e){System.out.println(e);}
19. t2.start();
20. t3.start();
21. }
22. }

Output:1

2

3

4

5

1

1

2

2

3

3

4

4

5

5

### **getName(),setName(String) and getId() method:**

|  |
| --- |
| public String getName() |
| public void setName(String name) |
| public long getId() |

1. **class** TestJoinMethod3 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestJoinMethod3 t1=**new** TestJoinMethod3();
7. TestJoinMethod3 t2=**new** TestJoinMethod3();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
10. System.out.println("id of t1:"+t1.getId());
12. t1.start();
13. t2.start();
15. t1.setName("Sonoo Jaiswal");
16. System.out.println("After changing name of t1:"+t1.getName());
17. }
18. }
19. Output:Name of t1:Thread-0
20. Name of t2:Thread-1
21. id of t1:8
22. running...
23. After changling name of t1:Sonoo Jaiswal
24. running...

## **Naming Thread**

The Thread class provides methods to change and get the name of a thread. By default, each thread has a name i.e. thread-0, thread-1 and so on. By we can change the name of the thread by using setName() method. The syntax of setName() and getName() methods are given below:

1. **public String getName():** is used to return the name of a thread.
2. **public void setName(String name):** is used to change the name of a thread.

## **Example of naming a thread**

1. **class** TestMultiNaming1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("running...");
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultiNaming1 t1=**new** TestMultiNaming1();
7. TestMultiNaming1 t2=**new** TestMultiNaming1();
8. System.out.println("Name of t1:"+t1.getName());
9. System.out.println("Name of t2:"+t2.getName());
11. t1.start();
12. t2.start();
14. t1.setName("Sonoo Jaiswal");
15. System.out.println("After changing name of t1:"+t1.getName());
16. }
17. }
18. Name of t1:Thread-0
19. Name of t2:Thread-1
20. id of t1:8
21. running...
22. After changeling name of t1:Sonoo Jaiswal
23. running...

# Priority of a Thread (Thread Priority):

|  |
| --- |
| Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. |

## **3 constants defined in Thread class:**

|  |
| --- |
| 1. public static int MIN\_PRIORITY 2. public static int NORM\_PRIORITY 3. public static int MAX\_PRIORITY |

# Daemon Thread in Java

**Daemon thread in java** is a service provider thread that provides services to the user thread. Its life depend on the mercy of user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

There are many java daemon threads running automatically e.g. gc, finalizer etc.

You can see all the detail by typing the jconsole in the command prompt. The jconsole tool provides information about the loaded classes, memory usage, running threads etc.

## **Points to remember for Daemon Thread in Java**

* It provides services to user threads for background supporting tasks. It has no role in life than to serve user threads.
* Its life depends on user threads.
* It is a low priority thread.

### **Why JVM terminates the daemon thread if there is no user thread?**

The sole purpose of the daemon thread is that it provides services to user thread for background supporting task. If there is no user thread, why should JVM keep running this thread. That is why JVM terminates the daemon thread if there is no user thread.

### **Methods for Java Daemon thread by Thread class**

The java.lang.Thread class provides two methods for java daemon thread.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public void setDaemon(boolean status) | is used to mark the current thread as daemon thread or user thread. |
| 2) | public boolean isDaemon() | is used to check that current is daemon. |

### **Simple example of Daemon thread in java**

*File: MyThread.java*

1. **public** **class** TestDaemonThread1 **extends** Thread{
2. **public** **void** run(){
3. **if**(Thread.currentThread().isDaemon()){//checking for daemon thread
4. System.out.println("daemon thread work");
5. }
6. **else**{
7. System.out.println("user thread work");
8. }
9. }
10. **public** **static** **void** main(String[] args){
11. TestDaemonThread1 t1=**new** TestDaemonThread1();//creating thread
12. TestDaemonThread1 t2=**new** TestDaemonThread1();
13. TestDaemonThread1 t3=**new** TestDaemonThread1();
15. t1.setDaemon(**true**);//now t1 is daemon thread
17. t1.start();//starting threads
18. t2.start();
19. t3.start();
20. }
21. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread1)

#### **Output**

daemon thread work

user thread work

user thread work

#### **Note: If you want to make a user thread as Daemon, it must not be started otherwise it will throw IllegalThreadStateException.**

*File: MyThread.java*

1. **class** TestDaemonThread2 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("Name: "+Thread.currentThread().getName());
4. System.out.println("Daemon: "+Thread.currentThread().isDaemon());
5. }
7. **public** **static** **void** main(String[] args){
8. TestDaemonThread2 t1=**new** TestDaemonThread2();
9. TestDaemonThread2 t2=**new** TestDaemonThread2();
10. t1.start();
11. t1.setDaemon(**true**);//will throw exception here
12. t2.start();
13. }
14. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestDaemonThread2)

Output:exception in thread main: java.lang.IllegalThreadStateException

# Java Thread Pool

**Java Thread pool** represents a group of worker threads that are waiting for the job and reuse many times.

In case of thread pool, a group of fixed size threads are created. A thread from the thread pool is pulled out and assigned a job by the service provider. After completion of the job, thread is contained in the thread pool again.

#### **Advantage of Java Thread Pool**

**Better performance** It saves time because there is no need to create new thread.

#### **Real time usage**

It is used in Servlet and JSP where container creates a thread pool to process the request.

#### **Example of Java Thread Pool**

Let's see a simple example of java thread pool using ExecutorService and Executors.

*File: WorkerThread.java*

1. **import** java.util.concurrent.ExecutorService;
2. **import** java.util.concurrent.Executors;
3. **class** WorkerThread **implements** Runnable {
4. **private** String message;
5. **public** WorkerThread(String s){
6. **this**.message=s;
7. }
8. **public** **void** run() {
9. System.out.println(Thread.currentThread().getName()+" (Start) message = "+message);
10. processmessage();//call processmessage method that sleeps the thread for 2 seconds
11. System.out.println(Thread.currentThread().getName()+" (End)");//prints thread name
12. }
13. **private** **void** processmessage() {
14. **try** {  Thread.sleep(2000);  } **catch** (InterruptedException e) { e.printStackTrace(); }
15. }
16. }

*File: JavaThreadPoolExample.java*

1. **public** **class** TestThreadPool {
2. **public** **static** **void** main(String[] args) {
3. ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads
4. **for** (**int** i = 0; i < 10; i++) {
5. Runnable worker = **new** WorkerThread("" + i);
6. executor.execute(worker);//calling execute method of ExecutorService
7. }
8. executor.shutdown();
9. **while** (!executor.isTerminated()) {   }
11. System.out.println("Finished all threads");
12. }
13. }

[download this example](https://www.javatpoint.com/src/multi/threadpool.zip)

Output:

pool-1-thread-1 (Start) message = 0

pool-1-thread-2 (Start) message = 1

pool-1-thread-3 (Start) message = 2

pool-1-thread-5 (Start) message = 4

pool-1-thread-4 (Start) message = 3

pool-1-thread-2 (End)

pool-1-thread-2 (Start) message = 5

pool-1-thread-1 (End)

pool-1-thread-1 (Start) message = 6

pool-1-thread-3 (End)

pool-1-thread-3 (Start) message = 7

pool-1-thread-4 (End)

pool-1-thread-4 (Start) message = 8

pool-1-thread-5 (End)

pool-1-thread-5 (Start) message = 9

pool-1-thread-2 (End)

pool-1-thread-1 (End)

pool-1-thread-4 (End)

pool-1-thread-3 (End)

pool-1-thread-5 (End)

Finished all threads

# How to perform single task by multiple threads?

|  |
| --- |
| If you have to perform single task by many threads, have only one run() method.For example: |

***Program of performing single task by multiple threads***

1. **class** TestMultitasking1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("task one");
4. }
5. **public** **static** **void** main(String args[]){
6. TestMultitasking1 t1=**new** TestMultitasking1();
7. TestMultitasking1 t2=**new** TestMultitasking1();
8. TestMultitasking1 t3=**new** TestMultitasking1();
10. t1.start();
11. t2.start();
12. t3.start();
13. }

### **How to perform multiple tasks by multiple threads (multitasking in multithreading)?**

|  |
| --- |
| If you have to perform multiple tasks by multiple threads,have multiple run() methods.For example: |

***Program of performing two tasks by two threads***

1. **class** Simple1 **extends** Thread{
2. **public** **void** run(){
3. System.out.println("task one");
4. }
5. }
7. **class** Simple2 **extends** Thread{
8. **public** **void** run(){
9. System.out.println("task two");
10. }
11. }
13. **class** TestMultitasking3{
14. **public** **static** **void** main(String args[]){
15. Simple1 t1=**new** Simple1();
16. Simple2 t2=**new** Simple2();
18. t1.start();
19. t2.start();
20. }
21. }

### **Same example as above by anonymous class that implements Runnable interface:**

***Program of performing two tasks by two threads***

1. **class** TestMultitasking5{
2. **public** **static** **void** main(String args[]){
3. Runnable r1=**new** Runnable(){
4. **public** **void** run(){
5. System.out.println("task one");
6. }
7. };
9. Runnable r2=**new** Runnable(){
10. **public** **void** run(){
11. System.out.println("task two");
12. }
13. };
15. Thread t1=**new** Thread(r1);
16. Thread t2=**new** Thread(r2);
18. t1.start();
19. t2.start();
20. }
21. }

**Java Runtime** class is used to interact with java runtime environment. Java Runtime class provides methods to execute a process, invoke GC, get total and free memory etc. There is only one instance of java.lang.Runtime class is available for one java application.

The **Runtime.getRuntime()** method returns the singleton instance of Runtime class.

## **Important methods of Java Runtime class**

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public static Runtime getRuntime() | returns the instance of Runtime class. |
| 2) | public void exit(int status) | terminates the current virtual machine. |
| 3) | public void addShutdownHook(Thread hook) | registers new hook thread. |
| 4) | public Process exec(String command)throws IOException | executes given command in a separate process. |
| 5) | public int availableProcessors() | returns no. of available processors. |
| 6) | public long freeMemory() | returns amount of free memory in JVM. |
| 7) | public long totalMemory() | returns amount of total memory in JVM. |

## **Java Runtime exec() method**

1. **public** **class** Runtime1{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("notepad");//will open a new notepad
4. }
5. }

## **How to shutdown system in Java**

You can use shutdown -s command to shutdown system. For windows OS, you need to provide full path of shutdown command e.g. c:\\Windows\\System32\\shutdown.

Here you can use -s switch to shutdown system, -r switch to restart system and -t switch to specify time delay.

1. **public** **class** Runtime2{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("shutdown -s -t 0");
4. }
5. }

## **How to shutdown windows system in Java**

1. **public** **class** Runtime2{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");
4. }
5. }

## **How to restart system in Java**

1. **public** **class** Runtime3{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime.getRuntime().exec("shutdown -r -t 0");
4. }
5. }

## **Java Runtime availableProcessors()**

1. **public** **class** Runtime4{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. System.out.println(Runtime.getRuntime().availableProcessors());
4. }
5. }

## **Java Runtime freeMemory() and totalMemory() method**

In the given program, after creating 10000 instance, free memory will be less than the previous free memory. But after gc() call, you will get more free memory.

1. **public** **class** MemoryTest{
2. **public** **static** **void** main(String args[])**throws** Exception{
3. Runtime r=Runtime.getRuntime();
4. System.out.println("Total Memory: "+r.totalMemory());
5. System.out.println("Free Memory: "+r.freeMemory());
7. **for**(**int** i=0;i<10000;i++){
8. **new** MemoryTest();
9. }
10. System.out.println("After creating 10000 instance, Free Memory: "+r.freeMemory());
11. System.gc();
12. System.out.println("After gc(), Free Memory: "+r.freeMemory());
13. }
14. }

Total Memory: 100139008

Free Memory: 99474824

After creating 10000 instance, Free Memory: 99310552

After gc(), Free Memory: 100182832

# Synchronization in Java

Synchronization in java is the capability to control the access of multiple threads to any shared resource.

Java Synchronization is better option where we want to allow only one thread to access the shared resource.

### **Why use Synchronization**

The synchronization is mainly used to

1. To prevent thread interference.
2. To prevent consistency problem.

### **Types of Synchronization**

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

### **Thread Synchronization**

There are two types of thread synchronization mutual exclusive and inter-thread communication.

1. Mutual Exclusive
   1. Synchronized method.
   2. Synchronized block.
   3. static synchronization.
2. Cooperation (Inter-thread communication in java)

### **Mutual Exclusive**

Mutual Exclusive helps keep threads from interfering with one another while sharing data. This can be done by three ways in java:

1. by synchronized method
2. by synchronized block
3. by static synchronization

### **Concept of Lock in Java**

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

From Java 5 the package java.util.concurrent.locks contains several lock implementations.

### **Understanding the problem without Synchronization**

In this example, there is no synchronization, so output is inconsistent. Let's see the example:

1. **class** Table{
2. **void** printTable(**int** n){//method not synchronized
3. **for**(**int** i=1;i<=5;i++){
4. System.out.println(n\*i);
5. **try**{
6. Thread.sleep(400);
7. }**catch**(Exception e){System.out.println(e);}
8. }
10. }
11. }
13. **class** MyThread1 **extends** Thread{
14. Table t;
15. MyThread1(Table t){
16. **this**.t=t;
17. }
18. **public** **void** run(){
19. t.printTable(5);
20. }
22. }
23. **class** MyThread2 **extends** Thread{
24. Table t;
25. MyThread2(Table t){
26. **this**.t=t;
27. }
28. **public** **void** run(){
29. t.printTable(100);
30. }
31. }
33. **class** TestSynchronization1{
34. **public** **static** **void** main(String args[]){
35. Table obj = **new** Table();//only one object
36. MyThread1 t1=**new** MyThread1(obj);
37. MyThread2 t2=**new** MyThread2(obj);
38. t1.start();
39. t2.start();
40. }
41. }

Output: 5

100

10

200

15

300

20

400

25

500

### **Java synchronized method**

If you declare any method as synchronized, it is known as synchronized method.

Synchronized method is used to lock an object for any shared resource.

When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

1. //example of java synchronized method
2. **class** Table{
3. **synchronized** **void** printTable(**int** n){//synchronized method
4. **for**(**int** i=1;i<=5;i++){
5. System.out.println(n\*i);
6. **try**{
7. Thread.sleep(400);
8. }**catch**(Exception e){System.out.println(e);}
9. }
11. }
12. }
14. **class** MyThread1 **extends** Thread{
15. Table t;
16. MyThread1(Table t){
17. **this**.t=t;
18. }
19. **public** **void** run(){
20. t.printTable(5);
21. }
23. }
24. **class** MyThread2 **extends** Thread{
25. Table t;
26. MyThread2(Table t){
27. **this**.t=t;
28. }
29. **public** **void** run(){
30. t.printTable(100);
31. }
32. }
34. **public** **class** TestSynchronization2{
35. **public** **static** **void** main(String args[]){
36. Table obj = **new** Table();//only one object
37. MyThread1 t1=**new** MyThread1(obj);
38. MyThread2 t2=**new** MyThread2(obj);
39. t1.start();
40. t2.start();
41. }
42. }

Output: 5

10

15

20

25

100

200

300

400

500

# Synchronized block in java

Synchronized block can be used to perform synchronization on any specific resource of the method.

Suppose you have 50 lines of code in your method, but you want to synchronize only 5 lines, you can use synchronized block.

If you put all the codes of the method in the synchronized block, it will work same as the synchronized method.

### Points to remember for Synchronized block

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.
* **class** Table{
* **void** printTable(**int** n){
* **synchronized**(**this**){//synchronized block
* **for**(**int** i=1;i<=5;i++){
* System.out.println(n\*i);
* **try**{
* Thread.sleep(400);
* }**catch**(Exception e){System.out.println(e);}
* }
* }
* }//end of the method
* }
* **public** **class** TestSynchronizedBlock2{
* **public** **static** **void** main(String args[]){
* **final** Table obj = **new** Table();//only one object
* Thread t1=**new** Thread(){
* **public** **void** run(){
* obj.printTable(5);
* }
* };
* Thread t2=**new** Thread(){
* **public** **void** run(){
* obj.printTable(100);
* }
* };
* t1.start();
* t2.start();
* }
* }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestSynchronizedBlock2)

Output:5

10

15

20

25

100

200

300

400

500

### **Problem without static synchronization**

Suppose there are two objects of a shared class(e.g. Table) named object1 and object2.In case of synchronized method and synchronized block there cannot be interference between t1 and t2 or t3 and t4 because t1 and t2 both refers to a common object that have a single lock.But there can be interference between t1 and t3 or t2 and t4 because t1 acquires another lock and t3 acquires another lock.I want no interference between t1 and t3 or t2 and t4.Static synchronization solves this problem.

class Table{

synchronized void printTable(int n){

for(int i=1;i<=10;i++){

System.out.println(n\*i);

try{

Thread.sleep(400);

}catch(Exception e){}

}

}

}

class MyThread1 extends Thread{

public void run(){

Table t = new Table();

t.printTable(1);

}

}

class MyThread2 extends Thread{

public void run(){

Table t = new Table();

t.printTable(10);

}

}

class MyThread3 extends Thread{

public void run(){

Table t = new Table();

t.printTable(100);

}

}

class MyThread4 extends Thread{

public void run(){

Table t = new Table();

t.printTable(1000);

}

}

public class TestSynchronization4{

public static void main(String t[]){

MyThread1 t1=new MyThread1();

MyThread2 t2=new MyThread2();

MyThread3 t3=new MyThread3();

MyThread4 t4=new MyThread4();

t1.start();

t2.start();

t3.start();

t4.start();

}

}

100  
1000  
1  
10  
200  
2000  
2  
20  
300  
3000  
3  
30  
400  
4000  
40  
4  
500  
5000  
50  
5  
600  
6000  
60  
6  
700  
7000  
70  
7  
800

Deadlock in java is a part of multithreading. Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.

1. **public** **class** TestDeadlockExample1 {
2. **public** **static** **void** main(String[] args) {
3. **final** String resource1 = "ratan jaiswal";
4. **final** String resource2 = "vimal jaiswal";
5. // t1 tries to lock resource1 then resource2
6. Thread t1 = **new** Thread() {
7. **public** **void** run() {
8. **synchronized** (resource1) {
9. System.out.println("Thread 1: locked resource 1");
11. **try** { Thread.sleep(100);} **catch** (Exception e) {}
13. **synchronized** (resource2) {
14. System.out.println("Thread 1: locked resource 2");
15. }
16. }
17. }
18. };
20. // t2 tries to lock resource2 then resource1
21. Thread t2 = **new** Thread() {
22. **public** **void** run() {
23. **synchronized** (resource2) {
24. System.out.println("Thread 2: locked resource 2");
26. **try** { Thread.sleep(100);} **catch** (Exception e) {}
28. **synchronized** (resource1) {
29. System.out.println("Thread 2: locked resource 1");
30. }
31. }
32. }
33. };

36. t1.start();
37. t2.start();
38. }
39. }

Output: Thread 1: locked resource 1

Thread 2: locked resource 2

### **1) wait() method**

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

|  |  |
| --- | --- |
| public final void wait()throws InterruptedException | waits until object is notified. |
| public final void wait(long timeout)throws InterruptedException |  |

### **2) notify() method**

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

public final void notify()

### **3) notifyAll() method**

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

|  |  |
| --- | --- |
| **wait()** | **sleep()** |
| wait() method releases the lock | sleep() method doesn't release the lock. |
| is the method of Object class | is the method of Thread class |
| is the non-static method | is the static method |
| is the non-static method | is the static method |
| should be notified by notify() or notifyAll() methods | after the specified amount of time, sleep is completed. |

### **Example of inter thread communication in java**

Let's see the simple example of inter thread communication.

1. **class** Customer{
2. **int** amount=10000;
4. **synchronized** **void** withdraw(**int** amount){
5. System.out.println("going to withdraw...");
7. **if**(**this**.amount<amount){
8. System.out.println("Less balance; waiting for deposit...");
9. **try**{wait();}**catch**(Exception e){}
10. }
11. **this**.amount-=amount;
12. System.out.println("withdraw completed...");
13. }
15. **synchronized** **void** deposit(**int** amount){
16. System.out.println("going to deposit...");
17. **this**.amount+=amount;
18. System.out.println("deposit completed... ");
19. notify();
20. }
21. }
23. **class** Test{
24. **public** **static** **void** main(String args[]){
25. **final** Customer c=**new** Customer();
26. **new** Thread(){
27. **public** **void** run(){c.withdraw(15000);}
28. }.start();
29. **new** Thread(){
30. **public** **void** run(){c.deposit(10000);}
31. }.start();
33. }}

Output: going to withdraw...

Less balance; waiting for deposit...

going to deposit...

deposit completed...

withdraw completed

1. **lass** TestInterruptingThread1 **extends** Thread{
2. **public** **void** run(){
3. **try**{
4. Thread.sleep(1000);
5. System.out.println("task");
6. }**catch**(InterruptedException e){
7. **throw** **new** RuntimeException("Thread interrupted..."+e);
8. }
10. }
12. **public** **static** **void** main(String args[]){
13. TestInterruptingThread1 t1=**new** TestInterruptingThread1();
14. t1.start();
15. **try**{
16. t1.interrupt();
17. }**catch**(Exception e){System.out.println("Exception handled "+e);}
19. }
20. }

Output:Exception in thread-0

java.lang.RuntimeException: Thread interrupted...

java.lang.InterruptedException: sleep interrupted

at A.run(A.java:7)

|  |
| --- |
| The isInterrupted() method returns the interrupted flag either true or false. The static interrupted() method returns the interrupted flag afterthat it sets the flag to false if it is true. |

1. **public** **class** TestInterruptingThread4 **extends** Thread{
3. **public** **void** run(){
4. **for**(**int** i=1;i<=2;i++){
5. **if**(Thread.interrupted()){
6. System.out.println("code for interrupted thread");
7. }
8. **else**{
9. System.out.println("code for normal thread");
10. }
12. }//end of for loop
13. }
15. **public** **static** **void** main(String args[]){
17. TestInterruptingThread4 t1=**new** TestInterruptingThread4();
18. TestInterruptingThread4 t2=**new** TestInterruptingThread4();
20. t1.start();
21. t1.interrupt();
23. t2.start();
25. }
26. }

Output:Code for interrupted thread

code for normal thread

code for normal thread

code for normal thread

Do You Know?

1. How to create generic class and generic method in java ?
2. What is annotation and how to create custom annotation ?
3. What is the advantage of assertion and where we should not use it ?
4. What is variable argument and what rules are defined for variable argument ?
5. What is the difference between import and static import ?
6. How autoboxing is applied in method overloading. Which concept beats autoboxing ?
7. What is enum type and how to specify specific value to the enum constants ?

#### [**JavaSE 8 Features**](https://www.javatpoint.com/java-8-features)

The important features of JavaSE 8 are lambda expressions, methods references, default methods, functional interface, java 8 date/time, stream classes etc.

* [Java 8 Date/Time API (Java 8)](https://www.javatpoint.com/java-date)
* [Lambda Expressions (Java 8)](https://www.javatpoint.com/java-lambda-expressions)
* [Method References (Java 8)](https://www.javatpoint.com/java-8-method-reference)
* [Functional Interfaces (Java 8)](https://www.javatpoint.com/java-8-functional-interfaces)
* [Stream (Java 8)](https://www.javatpoint.com/java-8-stream)
* [Base64 Encode Decode (Java 8)](https://www.javatpoint.com/java-base64-encode-decode)
* [Default Methods (Java 8)](https://www.javatpoint.com/java-default-methods)
* [forEach method(Java 8)](https://www.javatpoint.com/java-8-foreach)
* [Collectors(Java 8)](https://www.javatpoint.com/java-8-collectors)
* [StringJoiner(Java 8)](https://www.javatpoint.com/java-stringjoiner)
* [Optional class (Java 8)](https://www.javatpoint.com/java-8-optional)
* [Nashorn JavaScript (Java 8)](https://www.javatpoint.com/java-nashorn)
* [Parallel Array Sorting (Java 8)](https://www.javatpoint.com/java-8-parallel-array-sorting)
* [Type Inference (Java 8)](https://www.javatpoint.com/java-8-type-inference)
* [Method Parameter Reflection (Java 8)](https://www.javatpoint.com/java-8-method-parameter-reflection)
* [Type annotations and repeating annotations (Java 8)](https://www.javatpoint.com/java-8-type-annotations-and-repeating-annotations)
* [Java JDBC Improvements (Java 8)](https://www.javatpoint.com/java-8-jdbc-improvements)
* Java IO Improvement (Java 8)
* Java Concurrency Improvement (Java 8)

#### **JavaSE 7 Features**

The important features of JavaSE 7 are try with resource, catching multiple exceptions etc.

* String in switch statement (Java 7)
* Binary Literals (Java 7)
* The try-with-resources (Java 7)
* Caching Multiple Exceptions by single catch (Java 7)
* Underscores in Numeric Literals (Java 7)

#### **JavaSE 6 Features**

The important feature of JavaSE 6 is premain method (also known as instrumentation).

* Instrumentation (premain method) (Java 6)

#### **J2SE 5 Features**

The important features of J2SE 5 are generics and assertions. Others are auto-boxing, enum, var-args, static import, for-each loop (enhanced for loop etc.

* [For-each loop (Java 5)](https://www.javatpoint.com/for-each-loop)
* [Varargs (Java 5)](https://www.javatpoint.com/varargs)
* [Static Import (Java 5)](https://www.javatpoint.com/static-import-in-java)
* [Autoboxing and Unboxing (Java 5)](https://www.javatpoint.com/autoboxing-and-unboxing)
* [Enum (Java 5)](https://www.javatpoint.com/enum-in-java)
* [Covariant Return Type (Java 5)](https://www.javatpoint.com/covariant-return-type)
* [Annotation (Java 5)](https://www.javatpoint.com/java-annotation)
* [Generics (Java 5)](https://www.javatpoint.com/generics-in-java)

#### **J2SE 4 Features**

The important feature of J2SE 4 is assertions. It is used for testing.

* [Assertion](https://www.javatpoint.com/assertion-in-java) (Java 4)

Before generics, we can store any type of objects in collection i.e. non-generic. Now generics, forces the java programmer to store specific type of objects.

#### **Advantage of Java Generics**

There are mainly 3 advantages of generics. They are as follows:

**1) Type-safety :** We can hold only a single type of objects in generics. It doesn’t allow to store other objects.

**2) Type casting is not required:** There is no need to typecast the object.

Before Generics, we need to type cast.

1. List list = **new** ArrayList();
2. list.add("hello");
3. String s = (String) list.get(0);//typecasting

After Generics, we don't need to typecast the object.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. String s = list.get(0);

**3) Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.

1. List<String> list = **new** ArrayList<String>();
2. list.add("hello");
3. list.add(32);//Compile Time Error

## **Example of Java Generics using Map**

Now we are going to use map elements using generics. Here, we need to pass key and value. Let us understand it by a simple example:

1. **import** java.util.\*;
2. **class** TestGenerics2{
3. **public** **static** **void** main(String args[]){
4. Map<Integer,String> map=**new** HashMap<Integer,String>();
5. map.put(1,"vijay");
6. map.put(4,"umesh");
7. map.put(2,"ankit");
9. //Now use Map.Entry for Set and Iterator
10. Set<Map.Entry<Integer,String>> set=map.entrySet();
12. Iterator<Map.Entry<Integer,String>> itr=set.iterator();
13. **while**(itr.hasNext()){
14. Map.Entry e=itr.next();//no need to typecast
15. System.out.println(e.getKey()+" "+e.getValue());
16. }
18. }}

### **Abstraction in Java**

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

Abstraction lets you focus on what the object does instead of how it does it.

### **Ways to achieve Abstraction**

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

### **Abstract class in Java**

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

#### **Points to Remember**

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.
* **abstract** **class** Shape{
* **abstract** **void** draw();
* }
* //In real scenario, implementation is provided by others i.e. unknown by end user
* **class** Rectangle **extends** Shape{
* **void** draw(){System.out.println("drawing rectangle");}
* }
* **class** Circle1 **extends** Shape{
* **void** draw(){System.out.println("drawing circle");}
* }
* //In real scenario, method is called by programmer or user
* **class** TestAbstraction1{
* **public** **static** **void** main(String args[]){
* Shape s=**new** Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
* s.draw();
* }
* }

drawing circle

1. **abstract** **class** Bike{
2. Bike(){System.out.println("bike is created");}
3. **abstract** **void** run();
4. **void** changeGear(){System.out.println("gear changed");}
5. }
6. //Creating a Child class which inherits Abstract class
7. **class** Honda **extends** Bike{
8. **void** run(){System.out.println("running safely..");}
9. }
10. //Creating a Test class which calls abstract and non-abstract methods
11. **class** TestAbstraction2{
12. **public** **static** **void** main(String args[]){
13. Bike obj = **new** Honda();
14. obj.run();
15. obj.changeGear();
16. }
17. }
18. bike is created
19. running safely..
20. gear changed
21. **interface** A{
22. **void** a();
23. **void** b();
24. **void** c();
25. **void** d();
26. }
28. **abstract** **class** B **implements** A{
29. **public** **void** c(){System.out.println("I am c");}
30. }
32. **class** M **extends** B{
33. **public** **void** a(){System.out.println("I am a");}
34. **public** **void** b(){System.out.println("I am b");}
35. **public** **void** d(){System.out.println("I am d");}
36. }
38. **class** Test5{
39. **public** **static** **void** main(String args[]){
40. A a=**new** M();
41. a.a();
42. a.b();
43. a.c();
44. a.d();
45. }}

### **Methods of Java ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| void clear() | It is used to remove all of the elements from this list. |
| int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. |
| Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. |
| Object[] toArray(Object[] a) | It is used to return an array containing all of the elements in this list in the correct order. |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean addAll(int index, Collection c) | It is used to insert all of the elements in the specified collection into this list, starting at the specified position. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

### **Iterating Collection through for-each loop**

1. **import** java.util.\*;
2. **class** TestCollection2{
3. **public** **static** **void** main(String args[]){
4. ArrayList<String> al=**new** ArrayList<String>();
5. al.add("Ravi");
6. al.add("Vijay");
7. al.add("Ravi");
8. al.add("Ajay");
9. **for**(String obj:al)
10. System.out.println(obj);
11. }
12. }

### **User-defined class objects in Java ArrayList**

Let's see an example where we are storing Student class object in array list.

1. **class** Student{
2. **int** rollno;
3. String name;
4. **int** age;
5. Student(**int** rollno,String name,**int** age){
6. **this**.rollno=rollno;
7. **this**.name=name;
8. **this**.age=age;
9. }
10. }
11. **import** java.util.\*;
12. **public** **class** TestCollection3{
13. **public** **static** **void** main(String args[]){
14. //Creating user-defined class objects
15. Student s1=**new** Student(101,"Sonoo",23);
16. Student s2=**new** Student(102,"Ravi",21);
17. Student s2=**new** Student(103,"Hanumat",25);
18. //creating arraylist
19. ArrayList<Student> al=**new** ArrayList<Student>();
20. al.add(s1);//adding Student class object
21. al.add(s2);
22. al.add(s3);
23. //Getting Iterator
24. Iterator itr=al.iterator();
25. //traversing elements of ArrayList object
26. **while**(itr.hasNext()){
27. Student st=(Student)itr.next();
28. System.out.println(st.rollno+" "+st.name+" "+st.age);
29. }
30. }

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [**next →**](https://www.javatpoint.com/java-linkedlist)[**← prev**](https://www.javatpoint.com/collection-framework) Java ArrayList class Java ArrayList class hierarchy  Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.  The important points about Java ArrayList class are:   * Java ArrayList class can contain duplicate elements. * Java ArrayList class maintains insertion order. * Java ArrayList class is non synchronized. * Java ArrayList allows random access because array works at the index basis. * In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.  **Hierarchy of ArrayList class** As shown in above diagram, Java ArrayList class extends AbstractList class which implements List interface. The List interface extends Collection and Iterable interfaces in hierarchical order. **ArrayList class declaration** Let's see the declaration for java.util.ArrayList class.   1. **public** **class** ArrayList<E> **extends** AbstractList<E> **implements** List<E>, RandomAccess, Cloneable, Serializable  **Constructors of Java ArrayList**  |  |  | | --- | --- | | **Constructor** | **Description** | | ArrayList() | It is used to build an empty array list. | | ArrayList(Collection c) | It is used to build an array list that is initialized with the elements of the collection c. | | ArrayList(int capacity) | It is used to build an array list that has the specified initial capacity. |  **Methods of Java ArrayList**  |  |  | | --- | --- | | **Method** | **Description** | | void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. | | boolean addAll(Collection c) | It is used to append all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. | | void clear() | It is used to remove all of the elements from this list. | | int lastIndexOf(Object o) | It is used to return the index in this list of the last occurrence of the specified element, or -1 if the list does not contain this element. | | Object[] toArray() | It is used to return an array containing all of the elements in this list in the correct order. | | Object[] toArray(Object[] a) | It is used to return an array containing all of the elements in this list in the correct order. | | boolean add(Object o) | It is used to append the specified element to the end of a list. | | boolean addAll(int index, Collection c) | It is used to insert all of the elements in the specified collection into this list, starting at the specified position. | | Object clone() | It is used to return a shallow copy of an ArrayList. | | int indexOf(Object o) | It is used to return the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element. | | void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |    **Java Non-generic Vs Generic Collection** Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.  Java new generic collection allows you to have only one type of object in collection. Now it is type safe so typecasting is not required at run time.  Let's see the old non-generic example of creating java collection.   1. ArrayList al=**new** ArrayList();//creating old non-generic arraylist   Let's see the new generic example of creating java collection.   1. ArrayList<String> al=**new** ArrayList<String>();//creating new generic arraylist   In generic collection, we specify the type in angular braces. Now ArrayList is forced to have only specified type of objects in it. If you try to add another type of object, it gives *compile time error*. **Java ArrayList Example**  1. **import** java.util.\*; 2. **class** TestCollection1{ 3. **public** **static** **void** main(String args[]){ 4. ArrayList<String> list=**new** ArrayList<String>();//Creating arraylist 5. list.add("Ravi");//Adding object in arraylist 6. list.add("Vijay"); 7. list.add("Ravi"); 8. list.add("Ajay"); 9. //Traversing list through Iterator 10. Iterator itr=list.iterator(); 11. **while**(itr.hasNext()){ 12. System.out.println(itr.next()); 13. } 14. } 15. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection1)  Ravi  Vijay  Ravi  Ajay **Two ways to iterate the elements of collection in java** There are two ways to traverse collection elements:   1. By Iterator interface. 2. By for-each loop.   In the above example, we have seen traversing ArrayList by Iterator. Let's see the example to traverse ArrayList elements using for-each loop. **Iterating Collection through for-each loop**  1. **import** java.util.\*; 2. **class** TestCollection2{ 3. **public** **static** **void** main(String args[]){ 4. ArrayList<String> al=**new** ArrayList<String>(); 5. al.add("Ravi"); 6. al.add("Vijay"); 7. al.add("Ravi"); 8. al.add("Ajay"); 9. **for**(String obj:al) 10. System.out.println(obj); 11. } 12. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection2)  Ravi  Vijay  Ravi  Ajay **User-defined class objects in Java ArrayList** Let's see an example where we are storing Student class object in array list.   1. **class** Student{ 2. **int** rollno; 3. String name; 4. **int** age; 5. Student(**int** rollno,String name,**int** age){ 6. **this**.rollno=rollno; 7. **this**.name=name; 8. **this**.age=age; 9. } 10. } 11. **import** java.util.\*; 12. **public** **class** TestCollection3{ 13. **public** **static** **void** main(String args[]){ 14. //Creating user-defined class objects 15. Student s1=**new** Student(101,"Sonoo",23); 16. Student s2=**new** Student(102,"Ravi",21); 17. Student s2=**new** Student(103,"Hanumat",25); 18. //creating arraylist 19. ArrayList<Student> al=**new** ArrayList<Student>(); 20. al.add(s1);//adding Student class object 21. al.add(s2); 22. al.add(s3); 23. //Getting Iterator 24. Iterator itr=al.iterator(); 25. //traversing elements of ArrayList object 26. **while**(itr.hasNext()){ 27. Student st=(Student)itr.next(); 28. System.out.println(st.rollno+" "+st.name+" "+st.age); 29. } 30. } 31. }   [**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection3)  101 Sonoo 23  102 Ravi 21  103 Hanumat 25 **Example of addAll(Collection c) method**  1. **import** java.util.\*; 2. **class** TestCollection4{ 3. **public** **static** **void** main(String args[]){ 4. ArrayList<String> al=**new** ArrayList<String>(); 5. al.add("Ravi"); 6. al.add("Vijay"); 7. al.add("Ajay"); 8. ArrayList<String> al2=**new** ArrayList<String>(); 9. al2.add("Sonoo"); 10. al2.add("Hanumat"); 11. al.addAll(al2);//adding second list in first list 12. Iterator itr=al.iterator(); 13. **while**(itr.hasNext()){ 14. System.out.println(itr.next()); 15. } 16. } 17. }   **BOOK** |

1. **import** java.util.\*;
2. **class** Book {
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. }
14. **public** **class** ArrayListExample {
15. **public** **static** **void** main(String[] args) {
16. //Creating list of Books
17. List<Book> list=**new** ArrayList<Book>();
18. //Creating Books
19. Book b1=**new** Book(101,"Let us C","Yashwant Kanetkar","BPB",8);
20. Book b2=**new** Book(102,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
21. Book b3=**new** Book(103,"Operating System","Galvin","Wiley",6);
22. //Adding Books to list
23. list.add(b1);
24. list.add(b2);
25. list.add(b3);
26. //Traversing list
27. **for**(Book b:list){
28. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
29. }
30. }  5
31. }

### **LinkedList class declaration**

1. **public** **class** LinkedList<E> **extends** AbstractSequentialList<E> **implements** List<E>, Deque<E>, Cloneable, Serializable

### **Methods of Java LinkedList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, Object element) | It is used to insert the specified element at the specified position index in a list. |
| void addFirst(Object o) | It is used to insert the given element at the beginning of a list. |
| void addLast(Object o) | It is used to append the given element to the end of a list. |
| int size() | It is used to return the number of elements in a list |
| boolean add(Object o) | It is used to append the specified element to the end of a list. |
| boolean contains(Object o) | It is used to return true if the list contains a specified element. |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element in a list. |
| Object getFirst() | It is used to return the first element in a list. |
| Object getLast() | It is used to return the last element in a list. |
| int indexOf(Object o) | It is used to return the index in a list of the first occurrence of the specified element, or -1 if the list does not contain any element. |
| int lastIndexOf(Object o) | It is used to return the index in a list of the last occurrence of the specified element, or -1 if the list does not contain any element. |

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses **dynamic array** to store the elements. | LinkedList internally uses **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses array. If any element is removed from the array, all the bits are shifted in memory. | Manipulation with LinkedList is **faster** than ArrayList because it uses doubly linked list so no bit shifting is required in memory. |
| 3) ArrayList class can **act as a list** only because it implements List only. | LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces. |
| 4) ArrayList is **better for storing and accessing** data. | LinkedList is **better for manipulating** data. |

# Java List Interface

List Interface is the sub interface of Collection .It contains methods to insert and delete elements in index basis. It is a factory of ListIterator interface.

### **List Interface declaration**

1. **public** **interface** List<E> **extends** Collection<E>

### **Methods of Java List Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index,Object element) | It is used to insert element into the invoking list at the index passed in the index. |
| boolean addAll(int index,Collection c) | It is used to insert all elements of c into the invoking list at the index passed in the index. |
| object get(int index) | It is used to return the object stored at the specified index within the invoking collection. |
| object set(int index,Object element) | It is used to assign element to the location specified by index within the invoking list. |
| object remove(int index) | It is used to remove the element at position index from the invoking list and return the deleted element. |
| ListIterator listIterator() | It is used to return an iterator to the start of the invoking list. |
| ListIterator listIterator(int index) | It is used to return an iterator to the invoking list that begins at the specified index. |

## **Java ListIterator Interface**

ListIterator Interface is used to traverse the element in backward and forward direction.

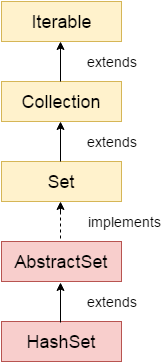
### **ListIterator Interface declaration**

1. **public** **interface** ListIterator<E> **extends** Iterator<E>

### **Methods of Java ListIterator Interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean hasNext() | This method return true if the list iterator has more elements when traversing the list in the forward direction. |
| Object next() | This method return the next element in the list and advances the cursor position. |
| boolean hasPrevious() | This method return true if this list iterator has more elements when traversing the list in the reverse direction. |
| Object previous() | This method return the previous element in the list and moves the cursor position backwards. |

**public** **interface** List<E> **extends** Collection<E>

Java HashSet class

Java HashSet class is used to create a collection that uses a hash table for storage. It inherits the AbstractSet class and implements Set interface.

The important points about Java HashSet class are:

* HashSet stores the elements by using a mechanism called **hashing.**
* HashSet contains unique elements only.

## **Difference between List and Set**

List can contain duplicate elements whereas Set contains unique elements only.

### **Hierarchy of HashSet class**

The HashSet class extends AbstractSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

1. **public** **class** HashSet<E> **extends** AbstractSet<E> **implements** Set<E>, Cloneable, Serializable

|  |  |  |
| --- | --- | --- |
| **SN** | **Constructor** | **Description** |
| 1) | HashSet() | It is used to construct a default HashSet. |
| 2) | HashSet(int capacity) | It is used to initialize the capacity of the hash set to the given integer value capacity. The capacity grows automatically as elements are added to the HashSet. |
| 3) | HashSet(int capacity, float loadFactor) | It is used to initialize the capacity of the hash set to the given integer value capacity and the specified load factor. |
| 4) | HashSet(Collection c) | It is used to initialize the hash set by using the elements of the collection c. |

## **Methods of Java HashSet class**

Various methods of Java HashSet class are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Method** | **Description** |
| 1) | boolean | [add(Object o)](https://www.javatpoint.com/java-hashset-add-method) | It is used to adds the specified element to this set if it is not already present. |
| 2) | void | [clear()](https://www.javatpoint.com/java-hashset-clear-method) | It is used to remove all of the elements from this set. |
| 3) | object | [clone()](https://www.javatpoint.com/java-hashset-clone-method) | It is used to return a shallow copy of this HashSet instance: the elements themselves are not cloned. |
| 4) | boolean | [contains(Object o)](https://www.javatpoint.com/java-hashset-contains-method) | It is used to return true if this set contains the specified element. |
| 5) | boolean | [isEmpty()](https://www.javatpoint.com/java-hashset-isempty-method) | It is used to return true if this set contains no elements. |
| 6) | Iterator<E> | [iterator()](https://www.javatpoint.com/java-hashset-iterator-method) | It is used to return an iterator over the elements in this set. |
| 7) | boolean | [remove(Object o)](https://www.javatpoint.com/java-hashset-remove-method) | It is used to remove the specified element from this set if it is present. |
| 8) | int | [size()](https://www.javatpoint.com/java-hashset-size-method) | It is used to return the number of elements in this set. |
| 9) | Spliterator<E> | [spliterator()](https://www.javatpoint.com/java-hashset-spliterator-method) | It is used to create a late-binding and fail-fast Spliterator over the elements in this set. |

**clear()** method

1. HashSet<String> hset = **new** HashSet<String>();
2. //Clear elements to hash set
3. hset.add("JTP");
4. hset.add("SSSIT");
5. hset.add("DFC");
6. //Checking elements in hash set
7. System.out.println("Hash set Elements: "+ hset);
8. //Clear set values
9. hset.clear();
10. System.out.println("Hash set elements after clear: "+ hset);

The **clone()** method of Java HashSet class is used to return a shallow copy of the specified HashSet.

**public** Object clone()

1. **public** **static** **void** main(String[] args) {
2. //Creating HashSet
3. HashSet<String> set = **new** HashSet<String>();
4. //Add elements into the Set
5. set.add("JavaTpoint");
6. set.add("Google");
7. set.add("Hindi100");
8. set.add("101");
9. set.add("Facebook");
10. //Print the elements
11. System.out.println("HashSet elements: " + set);
12. //Creating a new cloned set
13. HashSet<String> clonedSet = **new** HashSet<String>();
14. //Clone the HashSet
15. clonedSet = (HashSet)set.clone();
16. //Displaying the new Set after Cloning;
17. System.out.println("The new clone set elements: " + clonedSet);
18. }

HashSet elements: [Google, 101, Hindi100, JavaTpoint, Facebook]

The new clone set elements: [Google, 101, JavaTpoint, Facebook, Hindi100]

**contains()** method:

**public** **boolean** contains(Object o)

1. **public** **static** **void** main(String[] args) {
2. //Create hash set
3. HashSet<Integer> hset = **new** HashSet<Integer>();
4. //Add elements to hash set
5. hset.add(11);
6. hset.add(21);
7. hset.add(15);
8. hset.add(110);
9. hset.add(151);
10. //Print HashSet elements
11. System.out.println("Hash set Elements: "+ hset);
12. //Check for "110" in the set
13. System.out.println("Does the Set contains '110'? :- "+hset.contains(110));
14. //Check if the Set contains "555"
15. System.out.println("Does the Set contains '555'? :- "+hset.contains(555));
16. }
17. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=HashSetContainsExample1)

**Output:**

Hash set Elements: [21, 151, 11, 110, 15]

Does the Set contains '110'? :- true

Does the Set contains '555'? :- false

**isEmpty()** method:

**public** **boolean** isEmpty()

1. HashSet<String> hset = **new** HashSet<String>();
2. //Add elements into the Set
3. hset.add("Welcome");
4. hset.add("To");
5. hset.add("JavaTpoint");
6. //Print the HashSet Values
7. System.out.println("HashSet Elements: "+hset);
8. //Check for the empty set
9. System.out.println("Is the set empty: "+hset.isEmpty());
10. //Clearing the set and Check for the empty set
11. hset.clear();
12. System.out.println("Is the set empty: "+hset.isEmpty());

HashSet Elements: [Welcome, To, JavaTpoint]

Is the set empty: false

Is the set empty: true

1. **public** **class** HashSetIsEmptyExample3 {
2. **public** **static** **void** main(String[] args) {
3. //Get the HashMap object
4. HashSet studentSet = init();
5. //Validate the set
6. **if**(studentSet.isEmpty()){
7. System.out.println("student database is empty: "+studentSet);
8. }
9. **else**{
10. System.out.println("student database has "+studentSet.size()+" names: "+studentSet);
11. }
12. studentSet.clear();
13. **if**(studentSet.isEmpty()){
14. System.out.println("student database is empty: "+studentSet);
15. }
16. **else**{
17. System.out.println("student database has "+studentSet.size()+" names: "+studentSet);
18. }
19. }
20. **private** **static** HashSet init() {
21. //Create HashSet object
22. HashSet<String> studentSet = **new** HashSet<>();
23. //Add contents to the set
24. studentSet.add("Rahul");
25. studentSet.add("Mohan");
26. studentSet.add("Karan");
27. **return** studentSet;
28. }
29. }

student database has 3 names: [Rahul, Mohan, Karan]

student database is empty: []

**iterator()** method

**public** Iterator<E> iterator()

1. **public** **class** HashSetIteratorExample3 {
2. **public** **static** **void** main(String[] args) **throws** InterruptedException {
3. //Get the HashMap object
4. HashSet studentSet = init();
5. //Display the contents of set
6. Iterator<String> it = studentSet.iterator();
7. **while** (it.hasNext()) {
8. System.out.println(it.next());
9. }
10. }
11. **private** **static** HashSet init() {
12. //Create HashSet object
13. HashSet<String> studentSet = **new** HashSet<>();
14. //Add contents to the set
15. studentSet.add("Rahul");
16. studentSet.add("Mohan");
17. studentSet.add("Karan");
18. **return** studentSet;
19. }
20. }

**remove()** method:

1. **public** **boolean** remove(Object o)
2. **public** **class** HashSetRemoveExample1 {
3. **public** **static** **void** main(String[] args) {
4. HashSet<String> set=**new** HashSet<String>();
5. set.add("Ravi");
6. set.add("Vijay");
7. set.add("Aman");
8. set.add("Ajay");
9. //Display HashSet
10. System.out.println("HashSet: " + set);
11. //Remove elements using remove() method
12. set.remove("Vijay");
13. // Displaying the HashSet after removal
14. System.out.println("HashSet after removing elements: " + set);
15. }
16. }

**spliterator()** method of Java HashSet class is used to creates a late-binding and fail-fast Spliterator over the elements in this set.

1. Public Spliterator<E> spliterator()
2. **public** **class** HashSetSpliteratorExample3 {
3. **public** **static** **void** main(String[] args) {
4. List<String> list = Arrays.asList("Apple", "Banana", "Orange", "Graps");
5. Spliterator<String> s = list.spliterator();
6. s.tryAdvance(System.out::println);
7. System.out.println("--Bulk List traversal---");
8. s.forEachRemaining(System.out::println);
9. System.out.println(" --- Attempting tryAdvance again---");
10. **boolean** b = s.tryAdvance(System.out::println);
11. System.out.println("Element exists: "+b);
12. }
13. }

# 

# Java LinkedHashSet class

Java LinkedHashSet class is a Hash table and Linked list implementation of the set interface. It inherits HashSet class and implements Set interface.

The important points about Java LinkedHashSet class are:

* Contains unique elements only like HashSet.
* Provides all optional set operations, and permits null elements.
* Maintains insertion order.

## **Hierarchy of LinkedHashSet class**

The LinkedHashSet class extends HashSet class which implements Set interface. The Set interface inherits Collection and Iterable interfaces in hierarchical order.

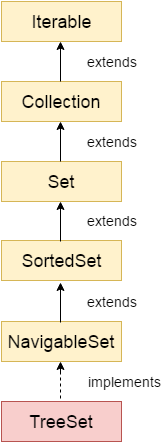
### **LinkedHashSet class declaration**

Let's see the declaration for java.util.LinkedHashSet class.

1. **public** **class** LinkedHashSet<E> **extends** HashSet<E> **implements** Set<E>, Cloneable, Ser

# Java HashSet class hierarchy

# Java TreeSet class

Java TreeSet class implements the Set interface that uses a tree for storage. It inherits AbstractSet class and implements NavigableSet interface. The objects of TreeSet class are stored in ascending order.

The important points about Java TreeSet class are:

* Contains unique elements only like HashSet.
* Access and retrieval times are quiet fast.
* Maintains ascending order.

### **Hierarchy of TreeSet class**

As shown in above diagram, Java TreeSet class implements NavigableSet interface. The NavigableSet interface extends SortedSet, Set, Collection and Iterable interfaces in hierarchical order.

### **TreeSet class declaration**

Let's see the declaration for java.util.TreeSet class.

1. **public** **class** TreeSet<E> **extends** AbstractSet<E> **implements** NavigableSet<E>, Cloneable, Serializable

### **Constructors of Java TreeSet class**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| TreeSet() | It is used to construct an empty tree set that will be sorted in an ascending order according to the natural order of the tree set. |
| TreeSet(Collection c) | It is used to build a new tree set that contains the elements of the collection c. |
| TreeSet(Comparator comp) | It is used to construct an empty tree set that will be sorted according to given comparator. |
| TreeSet(SortedSet ss) | It is used to build a TreeSet that contains the elements of the given SortedSet. |

### **Methods of Java TreeSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean addAll(Collection c) | It is used to add all of the elements in the specified collection to this set. |
| boolean contains(Object o) | It is used to return true if this set contains the specified element. |
| boolean isEmpty() | It is used to return true if this set contains no elements. |
| boolean remove(Object o) | It is used to remove the specified element from this set if it is present. |
| void add(Object o) | It is used to add the specified element to this set if it is not already present. |
| void clear() | It is used to remove all of the elements from this set. |
| Object clone() | It is used to return a shallow copy of this TreeSet instance. |
| Object first() | It is used to return the first (lowest) element currently in this sorted set. |
| Object last() | It is used to return the last (highest) element currently in this sorted set. |
| int size() | It is used to return the number of elements in this set. |

1. **import** java.util.\*;
2. **class** TestCollection11{
3. **public** **static** **void** main(String args[]){
4. //Creating and adding elements
5. TreeSet<String> al=**new** TreeSet<String>();
6. al.add("Ravi");
7. al.add("Vijay");
8. al.add("Ravi");
9. al.add("Ajay");
10. //Traversing elements
11. Iterator<String> itr=al.iterator();
12. **while**(itr.hasNext()){
13. System.out.println(itr.next());
14. }
15. }
16. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=TestCollection11)

Output:

Ajay

Ravi

Vijay

1. **import** java.util.\*;
2. **class** Book **implements** Comparable<Book>{
3. **int** id;
4. String name,author,publisher;
5. **int** quantity;
6. **public** Book(**int** id, String name, String author, String publisher, **int** quantity) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.author = author;
10. **this**.publisher = publisher;
11. **this**.quantity = quantity;
12. }
13. **public** **int** compareTo(Book b) {
14. **if**(id>b.id){
15. **return** 1;
16. }**else** **if**(id<b.id){
17. **return** -1;
18. }**else**{
19. **return** 0;
20. }
21. }
22. }
23. **public** **class** TreeSetExample {
24. **public** **static** **void** main(String[] args) {
25. Set<Book> set=**new** TreeSet<Book>();
26. //Creating Books
27. Book b1=**new** Book(121,"Let us C","Yashwant Kanetkar","BPB",8);
28. Book b2=**new** Book(233,"Operating System","Galvin","Wiley",6);
29. Book b3=**new** Book(101,"Data Communications & Networking","Forouzan","Mc Graw Hill",4);
30. //Adding Books to TreeSet
31. set.add(b1);
32. set.add(b2);
33. set.add(b3);
34. //Traversing TreeSet
35. **for**(Book b:set){
36. System.out.println(b.id+" "+b.name+" "+b.author+" "+b.publisher+" "+b.quantity);
37. }
38. }
39. }