Constructor

**Note:**  java compiler creates a default constructor if your class doesn't have any.

A Java constructor cannot be abstract, static, final, and synchronized

Note: We can use [access modifiers](https://www.javatpoint.com/access-modifiers) while declaring a constructor. It controls the object creation.

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another

Many ways to copy

By constructor, By assigning the values of one object into another, By clone() method of Object class

 //constructor to initialize another object

    Student6(Student6 s){

    id = s.id;

    name =s.name;

    }

Does constructor return any value?

Yes, it is the current class instance (You cannot use return type yet it returns a value).

Can constructor perform other tasks instead of initialization?

Yes, like object creation, starting a thread, calling a method, etc. You can perform any operation in the constructor as you perform in the method.

Is there Constructor class in Java?

Yes.

What is the purpose of Constructor class?

Java provides a Constructor class which can be used to get the internal information of a constructor in the class. It is found in the java.lang.reflect package.

Static

The **static keyword** in [Java](https://www.javatpoint.com/java-tutorial) is used for memory management mainly. We can apply static keyword with [variables](https://www.javatpoint.com/java-variables), methods, blocks and [nested classes](https://www.javatpoint.com/java-inner-class).

Java static variable gets memory only once in the class area at the time of class loading.

Advantages : makes your program **memory efficient** (i.e., it saves memory).

static method.

The static method can not use non static data member or call non-static method directly.

this and super cannot be used in static context.

Java static block

Is used to initialize the static data member.

It is executed before the main method at the time of classloading.

This

this() : to invoke current class constructor. it is used for constructor chaining. Call to this() must be the first statement

Inheritance

**IS-A relationship** which is also known as a parent-child relationship.

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Method Overloading is not possible by changing the return type of the method only.

One type is promoted to another implicitly if no matching datatype is found during method overloading.



Rules for Method Overriding

The method must have the same name, parameter as in the parent class

There must be an IS-A relationship (inheritance).

Can we override static method?

No, a static method cannot be overridden.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

Covariant Return Type

it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type.

**class** A

{

 A get(){**return** **this**;}

}

**class** B1 **extends** A

{

B1 get(){**return** **this**;}

**void** message(){System.out.println("welcome to covariant return type");}

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

{

n**ew** B1().get().message();

}

}

welcome to covariant return type

Advantages of Covariant Return Type

advantages of the covariant return type.

1) assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.

2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.

3) helps in preventing the run-time *ClassCastExceptions* on returns.

How is Covariant return types implemented?

Java doesn't allow the return type-based overloading, but JVM always allows return type-based overloading. JVM uses the full signature of a method for lookup/resolution. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.

Super Keyword in Java : It is a reference variable which is used to refer immediate parent class object.

super can be used to refer immediate parent class instance variable.

super can be used to invoke immediate parent class method, constructor

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().

**Instance initializer block**

|  |
| --- |
| **It** is used to initialize the instance data member. It run each time when object of the class is created. |
| Need : Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc. |

There are three places in java where you can perform operations: method, constructor, block

Rules: The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).The instance initializer block comes in the order in which they appear.

The **final keyword** in java is used to restrict the user. Final can be: variable, method, class

1) If you make any variable as final, you cannot change the value of final variable(It will be constant).

2) If you make any method as final, you cannot override it.

3) If you make any class as final, you cannot extend it.

Q) Is final method inherited? Yes, final method is inherited but you cannot override it.

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Q) What is final parameter? If you declare any parameter as final, you cannot change the value of it.

Q) Can we declare a constructor final? No, because constructor is never inherited.

If you overload a static method in Java, it is the example of compile time polymorphism.

**Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

In this process, an overridden method is called through the reference variable of a superclass. The determination of the method to be called is based on the object being referred to by the reference variable.

Rule: Runtime polymorphism can't be achieved by data members.

[Upcasting](https://www.geeksforgeeks.org/upcasting-vs-downcasting-in-java/) is casting a subtype to a super type in an upward direction to the inheritance tree.

[Downcasting](https://www.geeksforgeeks.org/upcasting-vs-downcasting-in-java/) refers to the procedure when subclass type refers to the object of the parent class is known as downcasting. The object which is already upcast, that object only can be performed downcast.

**Implementation:**

**(A)**Upcasting

|  |
| --- |
| class Parent  {    void show()    {      System.out.println("Parent show method is called");    }  }    class Child extends Parent    {      void show()      {      System.out.println("Child show method is called");      }    }    class Demo  {         Parent obj = new Child();         obj.show();      } |

**Output**

Child show method is called

**(B)**Downcasting

Diagram

Description automatically generated

|  |
| --- |
| class Vehicles {}    class Car extends Vehicles {      static void method(Vehicles v)      {          if (v instanceof Car) {              // Downcasting              Car c = (Car)v;                     System.out.println("Downcasting performed");          }      }        public static void main(String[] args)      {         Vehicles v = new Car();          Car.method(v);      }  } |

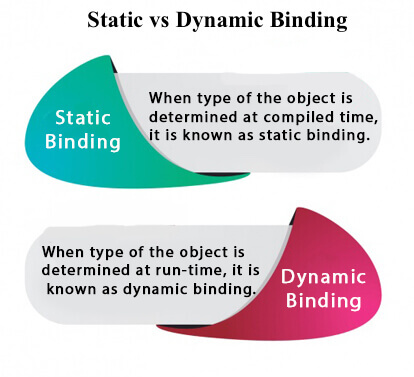
**Output**

Downcasting performed

*NOTE : Without perform upcast if we try to downcast , ClassCastException will be thrown.*

**Static Binding(Early Binding) and Dynamic Binding(Late Binding)**

Connecting a method call to the method body is known as binding.



When type of the object is determined at compiled time(by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example of static binding

**class** Dog{

**private** **void** eat(){System.out.println("dog is eating...");}

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

  Dog d1=**new** Dog();

  d1.eat();

 }

}

Dynamic binding

When type of the object is determined at run-time, it is known as dynamic binding.

Example of dynamic binding

**class** Animal{

**void** eat(){System.out.println("animal is eating...");}

}

**class** Dog **extends** Animal{

**void** eat(){System.out.println("dog is eating...");}

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

  Animal a=**new** Dog();

  a.eat();

 }

}

**Abstract class**

It can have [constructors](https://www.javatpoint.com/java-constructor) and static methods also.

It cannot be instantiated.

It can have final methods which will force the subclass not to change the body of the method.

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

Rule: If there is an abstract method in a class, that class must be abstract.

Rule: 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.

Another real scenario of abstract class

The abstract class can also be used to provide some implementation of the [interface](https://www.javatpoint.com/interface-in-java). In such case, the end user may not be forced to override all the methods of the interface.

**Interface in Java**

It has static constants and abstract methods. **represents the IS-A relationship**.

It is used to achieve abstraction. Support multiple inheritance. 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.

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.

Interface fields are public, static and final by default, and the methods are public and abstract.

A picture containing text, font, screenshot, businesscard

Description automatically generated

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

multiple inheritance is not supported in [class](https://www.javatpoint.com/object-and-class-in-java) because of ambiguity. Interface: no ambiguity;because its implementation is provided by the implementation class.

Java 8 Default Method in Interface

**interface** Drawable{

**default** **void** msg(){System.out.println("default method");}

}

Java 8 Static Method in Interface

**interface** Drawable{

**static** **int** cube(**int** x){**return** x\*x\*x;}

}

Q) What is marker or tagged interface?

An interface which has no member is known as a marker or tagged interface, for example, [Serializable](https://www.javatpoint.com/serialization-in-java), Cloneable, Remote, etc. They are used to provide some essential information to the JVM so that JVM may perform some useful operation.

//How Serializable interface is written?

**public** **interface** Serializable{

}

Nested Interface in Java

**interface** printable{

**void** print();

**interface** MessagePrintable{

**void** msg();

 }

}

**Java Package**

Advantage of Java Package

easily maintained; access protection; removes naming collision.

//save as Simple.java

**package** mypack;

**public** **class** Simple{

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

    System.out.println("Welcome to package");

   }

}

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

javac -d directory javafilename

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

|  |
| --- |
| **To Compile:** javac -d . Simple.java |
| **To Run:** java mypack.Simple |
|  |

How to access package from another package?

There are three ways to access the package from outside the package.

import package.\*;

import package.classname;

fully qualified name.

Note: If you import a package, subpackages will not be imported.

How to put two public classes in a package?

|  |
| --- |
| If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. |

**Static Import**

access any static member of a class directly.

**Access Modifiers**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Access Modifier** | **within class** | **within package** | **outside package by subclass only** | **outside package** |
| **Private** | Y | N | N | N |
| **Default** | Y | Y | N | N |
| **Protected** | Y | Y | Y | N |
| **Public** | Y | Y | Y | Y |

Note: A class cannot be private or protected except nested class.

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

The default modifier is more restrictive than protected.

Object Cloning in Java

The clone() method of Object class is used to clone an object.

The **java.lang.Cloneable interface** must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates **CloneNotSupportedException**.

The **clone() method** is defined in the Object class.

Why use clone() method ?

The **clone() method** saves the extra processing task for creating the exact copy of an object. If we perform it by using the new keyword, it will take a lot of processing time to be performed that is why we use object cloning.

Just define a parent class, implement Cloneable in it, provide the definition of the clone() method

Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

**class** Student18 **implements** Cloneable{

**public** Object clone()**throws** CloneNotSupportedException{

**return** **super**.clone();

}

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

**try**{

Student18 s1=**new** Student18(101,"amit");

Student18 s2=(Student18)s1.clone();

**}catch**(CloneNotSupportedException c){}

}

}

**Basic Math methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| [Math.abs()](https://www.javatpoint.com/java-math-abs-method) | It will return the Absolute value of the given value. |
| [Math.max()](https://www.javatpoint.com/java-math-max-method) | It returns the Largest of two values. |
| [Math.min()](https://www.javatpoint.com/java-math-min-method) | It is used to return the Smallest of two values. |
| [Math.round()](https://www.javatpoint.com/java-math-round-method) | It is used to round of the decimal numbers to the nearest value. |
| [Math.sqrt()](https://www.javatpoint.com/java-math-sqrt-method) | It is used to return the square root of a number. |
| [Math.cbrt()](https://www.javatpoint.com/java-math-cbrt-method) | It is used to return the cube root of a number. |
| [Math.pow()](https://www.javatpoint.com/java-math-pow-method) | It returns the value of first argument raised to the power to second argument. |
| [Math.signum()](https://www.javatpoint.com/java-math-signum-method) | It is used to find the sign of a given value. |
| [Math.ceil()](https://www.javatpoint.com/java-math-ceil-method) | It is used to find the smallest integer value that is greater than or equal to the argument or mathematical integer. |
| [Math.floor()](https://www.javatpoint.com/java-math-floor-method) | It is used to find the largest integer value which is less than or equal to the argument and is equal to the mathematical integer of a double value. |
| [Math.random()](https://www.javatpoint.com/java-math-random-method) | It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0. |

**Wrapper classes in Java**

Useul for Collections, Serialization, Synchronization, etc.

Autoboxing, unboxing and Creating the custom wrapper class

* **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it into objects through the wrapper classes.
* **Synchronization:** Java synchronization works with objects in Multithreading.
* **java.util package:** The java.util package provides the utility classes to deal with objects.
* **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

**Strictfp Keyword**

ensures that you will get the same result on every platform if you perform operations in the floating-point variable.

The strictfp keyword can be applied on methods, classes and interfaces.

The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

**Creating API Document | javadoc tool**

In the java file, we must use the documentation comment /\*\*... \*/ to post information for the class, method, constructor, fields etc.To create the document API, you need to use the javadoc tool followed by java file name. There is no need to compile the javafile.

On the command prompt, you need to write: javadoc M.java

to generate the document api. Now, there will be created a lot of html files. Open the index.html file to get the information about the classes.

**Java String**

Note: String objects are stored in a special memory area known as the "string constant pool".



Why Java uses the concept of String literal? To make Java more memory efficient

By new keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) 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).

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

 In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

Once String object is created its data or state can't be changed but a new String object is created.

There are three ways to compare String in Java:

By Using equals() Method

By Using == Operator

By compareTo() Method

1) By Using equals() Method : It compares values of string for equality.

2) By Using == operator : The == operator compares references not values.

3) By Using compareTo() method : compares values lexicographically

**s1 == s2** : The method returns 0.

**s1 > s2** : The method returns a positive value.

**s1 < s2** : The method returns a negative value.

**String Concatenation in Java**

 String s="Sachin"+" Tendulkar";

The **Java compiler transforms** above code to this:

String s=(**new** StringBuilder()).append("Sachin").append(" Tendulkar).toString();

Note: After a string literal, all the + will be treated as string concatenation operator.

There are some other possible ways to concatenate Strings in Java,

1. String concatenation using StringBuilder class

2. String concatenation using format() method

3. String concatenation using String.join() method (Java Version 8+)

4. String concatenation using StringJoiner class (Java Version 8+)

5. String concatenation using Collectors.joining() method (Java (Java Version 8+)

**Java StringBuffer**

Java StringBuffer class is used to create mutable (modifiable) String objects.

Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously.

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

What is a mutable String?

A String that can be modified or changed is known as mutable String. StringBuffer and StringBuilder classes are used for creating mutable strings.

Java StringBuilder Class

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.

String returns new hashcode while performing concatenation but the StringBuffer class returns same hashcode.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public StringBuilder append(String s) | It 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) | It 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) | It is used to replace the string from specified startIndex and endIndex. |
| public StringBuilder delete(int startIndex, int endIndex) | It is used to delete the string from specified startIndex and endIndex. |
| public StringBuilder reverse() | It is used to reverse the string. |
| public int capacity() | It is used to return the current capacity. |
| public void ensureCapacity(int minimumCapacity) | It is used to ensure the capacity at least equal to the given minimum. |
| public char charAt(int index) | It is used to return the character at the specified position. |
| public int length() | It is used to return the length of the string i.e. total number of characters. |
| public String substring(int beginIndex) | It is used to return the substring from the specified beginIndex. |
| public String substring(int beginIndex, int endIndex) | It is used to return the substring from the specified beginIndex and endIndex. |

|  |  |  |
| --- | --- | --- |
| **No.** | **String** | **StringBuffer** |
| 1) | The String class is immutable. | The StringBuffer class is mutable. |
| 2) | String is slow and consumes more memory when we concatenate too many strings because every time it creates new instance. | StringBuffer is fast and consumes less memory when we concatenate t 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. |
| 4) | String class is slower while performing concatenation operation. | StringBuffer class is faster while performing concatenation operation. |
| 5) | String class uses String constant pool. | StringBuffer uses Heap memory |

Difference between StringBuffer and StringBuilder

. The String class is an immutable class whereas StringBuffer and StringBuilder classes are mutable.

|  |  |  |
| --- | --- | --- |
| **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. |
| 3) | StringBuffer was introduced in Java 1.0 | StringBuilder was introduced in Java 1.5 |

How to create Immutable class?

There are many immutable classes like String, Boolean, Byte, Short, Integer, Long, Float, Double etc. In short, all the wrapper classes and String class is immutable. We can also create immutable class by creating final class that have final data members

* The instance variable of the class is final i.e. we cannot change the value of it after creating an object.
* The class is final so we cannot create the subclass.
* There is no setter methods i.e. we have no option to change the value of the instance variable.

mber: ABC123

Java String FAQs or Interview Questions

1) How many objects will be created in the following code?

String s1="javatpoint";

String s2="javatpoint";

**Answer:** Only one.

3) Is String class final?

**Answer:** Yes.

29.6MOOPs Concepts in Java

**Java Regex**

*define a pattern for searching or manipulating strings*.

The Matcher and Pattern classes provide the facility of Java regular expression.

**import** java.util.regex.\*;

**public** **class** RegexExample1{

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

//1st way

Pattern p = Pattern.compile(".s");//. represents single character

Matcher m = p.matcher("as");

**boolean** b = m.matches();

//2nd way

**boolean** b2=Pattern.compile(".s").matcher("as").matches();

//3rd way

**boolean** b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);

}}

Regex Character classes

|  |  |  |
| --- | --- | --- |
| **No.** | **Character Class** | **Description** |
| 1 | [abc] | a, b, or c (simple class) |
| 2 | [^abc] | Any character except a, b, or c (negation) |
| 3 | [a-zA-Z] | a through z or A through Z, inclusive (range) |
| 4 | [a-d[m-p]] | a through d, or m through p: [a-dm-p] (union) |
| 5 | [a-z&&[def]] | d, e, or f (intersection) |
| 6 | [a-z&&[^bc]] | a through z, except for b and c: [ad-z] (subtraction) |
| 7 | [a-z&&[^m-p]] | a through z, and not m through p: [a-lq-z](subtraction) |

Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all |
| X+ | X occurs once or more times |
| X\* | X occurs zero or more times |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times |
| X{y,z} | X occurs at least y times but less than z times |

Regex Metacharacters

The regular expression metacharacters work as shortcodes.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| . | Any character (may or may not match terminator) |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

**Exception Handling in Java**



There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception.

1) Checked Exception : The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. Checked exceptions are checked at compile-time.

2) Unchecked Exception : The classes that inherit the RuntimeException are known as unchecked exceptions. Are checked at runtime.

3) Error : Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError,

Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

**Java Exception Propagation**

An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method. If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.

Note: By default Unchecked Exceptions are forwarded in calling chain (propagated).

Note: By default, Checked Exceptions are not forwarded in calling chain (propagated).

Which exception should be declared?

**Ans:** Checked exception only, because:

**unchecked exception:** under our control so we can correct our code.

**error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

Checked Exception can be propagated (forwarded in call stack).

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

**There are two cases:**

**Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.

**Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Q) Can we rethrow an exception? Yes, by throwing same exception in catch block.

[**final**](https://www.javatpoint.com/final-keyword) is an access modifier, [**finally**](https://www.javatpoint.com/finally-block-in-exception-handling) is the block in Exception Handling and [**finalize**](https://www.javatpoint.com/java-object-finalize-method) is the method of object class.

|  |  |  |  |
| --- | --- | --- | --- |
| **Key** | **final** | **finally** | **finalize** |
| Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| Applicable to | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| Functionality | (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| Execution | Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the object is destroyed. |

Exception Handling with Method Overriding in Java

If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.

If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

**Java Custom Exception**

**public** **class** WrongFileNameException **extends** Exception {

**public** WrongFileNameException(String errorMessage) {

**super**(errorMessage);

    }

}

**Java Inner Classes (Nested Classes)**

To logically group classes and interfaces in one place to be more readable and maintainable.

Additionally, it can access all the members of the outer class, including private data members and methods.

Sometimes users need to program a class in such a way so that no other class can access it.

If all the class objects are a part of the outer object then it is easier to nest that class inside the outer class.

Non-static nested class (inner class)

|  |  |
| --- | --- |
| **Type** | **Description** |
| [Member Inner Class](https://www.javatpoint.com/member-inner-class) | A class created within class and outside method. |
| [Anonymous Inner Class](https://www.javatpoint.com/anonymous-inner-class) | A class created for implementing an interface or extending class. The java compiler decides its name. |
| [Local Inner Class](https://www.javatpoint.com/local-inner-class) | A class was created within the method. |
| [Static Nested Class](https://www.javatpoint.com/static-nested-class) | A static class was created within the class. |
| [Nested Interface](https://www.javatpoint.com/nested-interface) | An interface created within class or interface. |

Java Member Inner class

A non-static class that is created inside a class but outside a method is called **member inner class**. It is also known as a **regular inner class**. It can be declared with access modifiers like public, default, private, and protected.

**class** TestMemberOuter1{

**private** **int** data=30;

**class** Inner{

**void** msg(){System.out.println("data is "+data);}

 }

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

  TestMemberOuter1 obj=**new** TestMemberOuter1();

  TestMemberOuter1.Inner in=obj.**new** Inner();

  in.msg();

 }

}

Java anonymous inner class example using class

**abstract** **class** Person{

**abstract** **void** eat();

}

**class** TestAnonymousInner{

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

  Person p=**new** Person(){

**void** eat(){System.out.println("nice fruits");}

  };

  p.eat();

 }

}

Java anonymous inner class example using interface

**interface** Eatable

 Eatable e=**new** Eatable(){

**public** **void** eat(){System.out.println("nice fruits");}

 };

 e.eat();

Java Local inner class

A class i.e., created inside a method,block,forloop,if is called local inner class in java. cannot have any access modifiers associated with them. Can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

**public** **class** localInner1{

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

**void** display(){

**class** Local{

**void** msg(){System.out.println(data);}

  }

  Local l=**new** Local();

  l.msg();

 }

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

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

  obj.display();

 }

}

Java static nested class

is created inside a class. It cannot access non-static data members and methods. It can be accessed by outer class name.It can access static data members of the outer class, including private.

**class** TestOuter1{

**static** **int** data=30;

**static** **class** Inner{

**void** msg(){System.out.println("data is "+data);}

  }

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

  TestOuter1.Inner obj=**new** TestOuter1.Inner();

  obj.msg();

  }

}

Java Nested Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. must be referred to by the outer interface or class. It can't be accessed directly.

The nested interface must be public if it is declared inside the interface, but it can have any access modifier if declared within the class.

Nested interfaces are declared static

**interface** Showable{

**void** show();

**interface** Message{

**void** msg();

  }

}

**class** TestNestedInterface1 **implements** Showable.Message{

**public** **void** msg(){System.out.println("Hello nested interface");}

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

  Showable.Message message=**new** TestNestedInterface1();//upcasting here

  message.msg();

 }

}

Can we define a class inside the interface?

Yes, if we define a class inside the interface, the Java compiler creates a static nested class.

**Multithreading in Java**



1) Java Thread Example by extending Thread class

**class** Multi **extends** Thread{

**public** **void** run(){

System.out.println("thread is running...");

}

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

Multi t1=**new** Multi();

t1.start();

 }

}

2) Java Thread Example by implementing Runnable interface

**class** Multi3 **implements** Runnable{

**public** **void** run(){

System.out.println("thread is running...");

}

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

Multi3 m1=**new** Multi3();

Thread t1 =**new** Thread(m1);   // Using the constructor Thread(Runnable r)

t1.start();

 }

}

Important Points to Remember About the Sleep() Method

it always halts the execution of the current thread. Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown. If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

Can 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.

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

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

Naming Thread

By default, thread-0, thread-1. setName() and getName()

Current Thread

The currentThread() method returns a reference of the currently executing thread.

  System.out.println(Thread.currentThread().getName());

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.

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.

Difference between JDK, JRE, and JVM

Methods for Java Daemon thread by Thread class

Thread.currentThread().isDaemon()

t1.setDaemon(**true**);//now t1 is daemon thread

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

**Java Thread Pool**

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

Thread Pool Methods

**newFixedThreadPool(int s):** The method creates a thread pool of the fixed size s.

**newCachedThreadPool():** The method creates a new thread pool that creates the new threads when needed but will still use the previously created thread whenever they are available to use.

**newSingleThreadExecutor():** The method creates a new thread.

Advantage of Java Thread Pool

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

Real time usage

Example of Java Thread Pool

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

***File: WorkerThread.java***

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**class** WorkerThread **implements** Runnable {  }

**public** **class** TestThreadPool {

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

        ExecutorService executor = Executors.newFixedThreadPool(5);//creating a pool of 5 threads

**for** (**int** i = 0; i < 10; i++) {

            Runnable worker = **new** WorkerThread("" + i);

            executor.execute(worker);//calling execute method of ExecutorService

          }

        executor.shutdown();

**while** (!executor.isTerminated()) {   }

        System.out.println("Finished all threads");

    }

 }

**Risks involved in Thread Pools**

**Deadlock**

**Thread Leakage:** Leakage of threads occurs when a thread is being removed from the pool to execute a task but is not returning to it after the completion of the task.

**Resource Thrashing:** A lot of time is wasted in context switching among threads when the size of the thread pool is very large. Whenever there are more threads than the optimal number may cause the starvation problem, and it leads to resource thrashing.

Points to Remember

Do not queue the tasks that are concurrently waiting for the results obtained from the other tasks. It may lead to a deadlock situation, as explained above.

Care must be taken whenever threads are used for the operation that is long-lived. It may result in the waiting of thread forever and will finally lead to the leakage of the resource.

In the end, the thread pool has to be ended explicitly. If it does not happen, then the program continues to execute, and it never ends. Invoke the shutdown() method on the thread pool to terminate the executor. Note that if someone tries to send another task to the executor after shutdown, it will throw a RejectedExecutionException.

**Tuning the Thread Pool**

The accurate size of a thread pool is decided by the number of available processors and the type of tasks the threads have to execute. If a system has the P processors that have only got the computation type processes, then the maximum size of the thread pool of P or P + 1 achieves the maximum efficiency. However, the tasks may have to wait for I/O, and in such a scenario, one has to take into consideration the ratio of the waiting time (W) and the service time (S) for the request; resulting in the maximum size of the pool P \* (1 + W / S) for the maximum efficiency.

**ThreadGroup**

Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

Note: Now suspend(), resume() and stop() methods are deprecated.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Let's see a code to group multiple threads.

ThreadGroup tg1 = **new** ThreadGroup("Group A");

Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");

Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");

Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

Thread.currentThread().getThreadGroup().interrupt();

Java Shutdown Hook

In nutshell, the shutdown hook can be used to perform cleanup resources or save the state when JVM shuts down normally or abruptly. Performing clean resources means closing log files, sending some alerts, or something else. So if you want to execute some code before JVM shuts down, use the shutdown hook.

**class** MyThread **extends** Thread{

**public** **void** run(){

        System.out.println("shut down hook task completed..");

    }

}

**public** **class** TestShutdown1{

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

Runtime r=Runtime.getRuntime();

r.addShutdownHook(**new** MyThread());

System.out.println("Now main sleeping... press ctrl+c to exit");

**try**{Thread.sleep(3000);}**catch** (Exception e) {}

}

}

How to perform single task by multiple threads in Java?

**class** TestMultitasking1 **extends** Thread{

**public** **void** run(){

   System.out.println("task one");

 }

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

  TestMultitasking1 t1=**new** TestMultitasking1();

  TestMultitasking1 t2=**new** TestMultitasking1();

  TestMultitasking1 t3=**new** TestMultitasking1();

  t1.start();    t2.start();    t3.start();

 }

}

task one

Note: Each thread run in a separate callstack.

Printing even and odd numbers using two threads

To print the even and odd numbers using the two threads, we will use the synchronized block and the notify() method. Observe the following program.

**public** **class** OddEvenExample

{

**int** contr = 1;

**static** **int** NUM;

**public** **void** displayOddNumber()

{

**synchronized** (**this**)

{

**while** (contr < NUM)

{

**while** (contr % 2 == 0)

{

**try**

{

wait();

}

**catch** (InterruptedException ex)

{

ex.printStackTrace();

}

}

System.out.print(contr + " ");

contr = contr + 1;

notify();

}

}

}

**public** **void** displayEvenNumber()

{

…..

**while** (contr % 2 == 1)

…..

}

}

}

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

{

NUM = 20;

OddEvenExample oe = **new** OddEvenExample();

Thread th1 = **new** Thread(**new** Runnable()

{

**public** **void** run()

{

oe.displayEvenNumber();

}

});

Thread th2 = **new** Thread(**new** Runnable()

{

**public** **void** run()

{

oe.displayOddNumber();

}

});

th1.start();

th2.start();

}

}

**Java Garbage Collection**

In java, garbage means unreferenced objects.

How can an object be unreferenced?

By nulling the reference

By assigning a reference to another

By anonymous object : **new** Employee();

**finalize() method**

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing.

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.

Note: Neither finalization nor garbage collection is guaranteed.

**Java Runtime class**

**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.

  Runtime.getRuntime().exec("notepad");//will open a new notepad

**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.

  Runtime.getRuntime().exec("shutdown -s -t 0");

  Runtime.getRuntime().exec("c:\\Windows\\System32\\shutdown -s -t 0");

How to restart system in Java

  Runtime.getRuntime().exec("shutdown -r -t 0");

Java Runtime availableProcessors()

  System.out.println(Runtime.getRuntime().availableProcessors());

Java Runtime freeMemory() and totalMemory() method

  Runtime r=Runtime.getRuntime();

  System.out.println("Total Memory: "+r.totalMemory());

  System.out.println("Free Memory: "+r.freeMemory());

**Synchronization**

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

To prevent thread interference.

To prevent consistency problem.

Types of Synchronization

Process Synchronization

Thread Synchronization

**Thread Synchronization**

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

Mutual Exclusive : Synchronized method. Synchronized block. Static synchronization.

Cooperation (Inter-thread communication in java)

**Concept of Lock**

Synchronization is built around an internal entity known as the lock or monitor. Every object has a 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.

Java Synchronized Method

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

**synchronized** **void** printTable(**int** n){}//synchronized method

Synchronized Block in Java

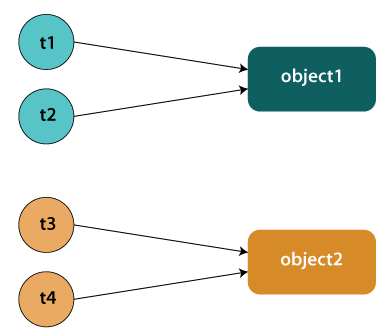
Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.

**synchronized**(**this**){//synchronized block

     }

Static Synchronization

If you make any static method as synchronized, the lock will be on the class not on object.



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. We don't want interference between t1 and t3 or t2 and t4. Static synchronization solves this problem.

Synchronized block on a class lock:

The block synchronizes on the lock of the object denoted by the reference .class name .class. A static synchronized method printTable(int n) in class Table is equivalent to the following declaration:

**static** **void** printTable(**int** n) {

**synchronized** (Table.**class**) {       // Synchronized block on class A

        // ...

    }

}

Deadlock in Java : threads are waiting for each other to release the lock, the condition is called deadlock.

More Complicated Deadlocks : A deadlock may also include more than two threads.

Deadlocks cannot be completely resolved but can be avoided

Avoid Nested Locks, Avoid Unnecessary Locks,

Using Thread Join: A deadlock usually happens when one thread is waiting for the other to finish

**Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**: wait(), notify(), notifyAll()

1) wait() method

The 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.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

SQL CREATE TABLE

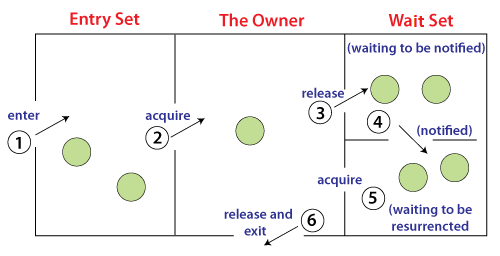
2) notify() method

The 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.

3) notifyAll() method

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

Understanding the process of inter-thread communication



Threads enter to acquire lock.

Lock is acquired by on thread.

Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.

If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).

Now thread is available to acquire lock.

After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

It is because they are related to lock and object has a lock.

Difference between wait and sleep?

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

Example of Inter Thread Communication in Java

**class** Customer

{

**int** amount=10000;

**synchronized** **void** withdraw(**int** amount)

{

System.out.println("going to withdraw...");

**if**(**this**.amount<amount){

System.out.println("Less balance; waiting for deposit...");

**try**{wait();}**catch**(Exception e){}

**this**.amount-=amount;

System.out.println("withdraw completed...");

}

**synchronized** **void** deposit(**int** amount)

{

System.out.println("going to deposit...");

**this**.amount+=amount;

System.out.println("deposit completed... ");

notify();

}

}

**class** Test

{

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

{

**final** Customer c=**new** Customer();

**new** Thread(){

**public** **void** run(){c.withdraw(15000);}

}.start();

**new** Thread(){

**public** **void** run(){c.deposit(10000);}

}.start();

}}

**Interrupting a Thread:**

If any thread is in sleeping or waiting state (i.e. sleep() or wait() is invoked), calling the interrupt() method on the thread, breaks out the sleeping or waiting state throwing InterruptedException. If the thread is not in the sleeping or waiting state, calling the interrupt() method performs normal behaviour and doesn't interrupt the thread but sets the interrupt flag to true. Let's first see the methods provided by the Thread class for thread interruption.

The 3 methods provided by the Thread class for interrupting a thread

public void interrupt()

public static boolean interrupted()

public boolean isInterrupted()

**Reentrant Monitor in Java**

Java monitors are reentrant means java thread can reuse the same monitor for different synchronized methods if method is called from the method.

Advantage of Reentrant Monitor

It eliminates the possibility of single thread deadlocking

**class** Reentrant {

**public** **synchronized** **void** m() {

    n();

    System.out.println("this is m() method");

    }

**public** **synchronized** **void** n() {

    System.out.println("this is n() method");

    }

}

**Serialization and Deserialization in Java**

serialization and deserialization process is platform-independent

For serializing the object, we call the **writeObject()** method of *ObjectOutputStream*class, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization

It is mainly used to travel object's state on the network (that is known as marshalling).

**import** java.io.\*;

**class** Persist{

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

**try**{

  //Creating the object

  Student s1 =**new** Student(211,"ravi");

  //Creating stream and writing the object

  FileOutputStream fout=**new** FileOutputStream("f.txt");

  ObjectOutputStream out=**new** ObjectOutputStream(fout);

  out.writeObject(s1);

  out.flush();

  //closing the stream

  out.close();

  System.out.println("success");

  }**catch**(Exception e){System.out.println(e);}

 }

}

**import** java.io.\*;

**class** Depersist{

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

**try**{

  //Creating stream to read the object

  ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt"));

  Student s=(Student)in.readObject();

  //printing the data of the serialized object

  System.out.println(s.id+" "+s.name);

  //closing the stream

  in.close();

  }**catch**(Exception e){System.out.println(e);}

 }

}

Java Serialization with Inheritance (IS-A Relationship)

If a class implements **Serializable interface** then all its sub classes will also be serializable.

Java Serialization with Aggregation (HAS-A Relationship)

If a class has a reference to another class, all the references must be Serializable otherwise serialization process will not be performed. In such case, *NotSerializableException* is thrown at runtime.

Java Serialization with the static data member

If there is any static data member in a class, it will not be serialized because static is the part of class not object.

Java Serialization with array or collection

Rule: In case of array or collection, all the objects of array or collection must be serializable. If any object is not serialiizable, serialization will be failed.

Externalizable in java

The Externalizable interface provides the facility of writing the state of an object into a byte stream in compress format. It is not a marker interface.

Java Transient Keyword

If you don't want to serialize any data member of a class, you can mark it as transient. Ex: user's login details and password

**transient** **int** id;

when you deserialize the object after serialization, you will not get the value of id. It will return default value

SerialVersionUID

The serialization process at runtime associates an id with each Serializable class. used to verify the sender and receiver of the serialized object.

**class** Employee **implements** Serializable{

**private** **static** **final** **long** serialVersionUID=1L;

**Collections**



**Methods of Collection interface**

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean add(E e) | It is used to insert an element in this collection. |
| 2 | public boolean addAll(Collection<? extends E> c) | It is used to insert the specified collection elements in the invoking collection. |
| 3 | public boolean remove(Object element) | It is used to delete an element from the collection. |
| 4 | public boolean removeAll(Collection<?> c) | It is used to delete all the elements of the specified collection from the invoking collection. |
| 5 | default boolean removeIf(Predicate<? super E> filter) | It is used to delete all the elements of the collection that satisfy the specified predicate. |
| 6 | public boolean retainAll(Collection<?> c) | It is used to delete all the elements of invoking collection except the specified collection. |
| 7 | public int size() | It returns the total number of elements in the collection. |
| 8 | public void clear() | It removes the total number of elements from the collection. |
| 9 | public boolean contains(Object element) | It is used to search an element. |
| 10 | public boolean containsAll(Collection<?> c) | It is used to search the specified collection in the collection. |
| 11 | public Iterator iterator() | It returns an iterator. |
| 12 | public Object[] toArray() | It converts collection into array. |
| 13 | public <T> T[] toArray(T[] a) | It converts collection into array. Here, the runtime type of the returned array is that of the specified array. |
| 14 | public boolean isEmpty() | It checks if collection is empty. |
| 15 | default Stream<E> parallelStream() | It returns a possibly parallel Stream with the collection as its source. |
| 16 | default Stream<E> stream() | It returns a sequential Stream with the collection as its source. |
| 17 | default Spliterator<E> spliterator() | It generates a Spliterator over the specified elements in the collection. |
| 18 | public boolean equals(Object element) | It matches two collections. |
| 19 | public int hashCode() | It returns the hash code number of the collection. |

**Iterator interface**

|  |
| --- |
| Iterator interface provides the facility of iterating the elements in a forward direction only. |

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | public boolean hasNext() | It returns true if the iterator has more elements otherwise it returns false. |
| 2 | public Object next() | It returns the element and moves the cursor pointer to the next element. |
| 3 | public void remove() | It removes the last elements returned by the iterator. It is less used. |

**ArrayList**

Java **ArrayList** class uses a *dynamic array*[*;*](https://www.javatpoint.com/array-in-java) there is *no size limit*;can have the duplicate elements also;maintains the insertion order internally;is non [synchronized](https://www.javatpoint.com/synchronization-in-java)

allows random access because array works at the index basis.

manipulation is little bit slower than the LinkedList in Java because a lot of shifting needs to occur if any element is removed from the array list.

  ArrayList<String> list=**new** ArrayList<String>();

 list.add("Apple");

Iterator itr=list.iterator();//getting the Iterator

**while**(itr.hasNext()){//check if iterator has the elements

   System.out.println(itr.next());//printing the element and move to next

 //Sorting the list

  Collections.sort(list2);

**Methods of ArrayList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void [add](https://www.javatpoint.com/java-arraylist-add-method)(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean [addAll](https://www.javatpoint.com/java-arraylist-addall-method)(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void ensureCapacity(int requiredCapacity) | It is used to enhance the capacity of an ArrayList instance. |
| E get(int index) | It is used to fetch the element from the particular position of the list. |
| [listIterator()](https://www.javatpoint.com/java-arraylist-listiterator-method) |  |
| 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 clone() | It is used to return a shallow copy of an ArrayList. |
| boolean contains(Object o) | It returns true if the list contains the specified element |
| 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. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |
| boolean removeIf(Predicate<? super E> filter) | It is used to remove all the elements from the list that satisfies the given predicate. |
| protected void [removeRange](https://www.javatpoint.com/java-arraylist-removerange-method)(int fromIndex, int toIndex) | It is used to remove all the elements lies within the given range. |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of specified comparator. |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |
| void trimToSize() | It is used to trim the capacity of this ArrayList instance to be the list's current size. |

**Java LinkedList**

 LinkedList<String> al=**new** LinkedList<String>();

**Methods of Java LinkedList**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, E element) | It is used to insert the specified element at the specified position index in a list. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| void addFirst(E e) | It is used to insert the given element at the beginning of a list. |
| void addLast(E e) | It is used to append the given element to the end of a list. |
| Object clone() | It is used to return a shallow copy of an ArrayList. |
| Iterator<E> descendingIterator() | It is used to return an iterator over the elements in a deque in reverse sequential order. |
| E element() | It is used to retrieve the first element of a list. |
| E get(int index) | It is used to return the element at the specified position in a list. |
| E getFirst() | It is used to return the first element in a list. |
| E 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. |
| ListIterator<E> listIterator(int index) | It is used to return a list-iterator of the elements in proper sequence, starting at the specified position in the list. |
| boolean offer(E e) | It adds the specified element as the last element of a list. |
| boolean offerFirst(E e) | It inserts the specified element at the front of a list. |
| boolean offerLast(E e) | It inserts the specified element at the end of a list. |
| E peek() | It retrieves the first element of a list |
| E peekFirst() | It retrieves the first element of a list or returns null if a list is empty. |
| E peekLast() | It retrieves the last element of a list or returns null if a list is empty. |
| E poll() | It retrieves and removes the first element of a list. |
| E pollFirst() | It retrieves and removes the first element of a list, or returns null if a list is empty. |
| E pollLast() | It retrieves and removes the last element of a list, or returns null if a list is empty. |
| E pop() | It pops an element from the stack represented by a list. |
| void push(E e) | It pushes an element onto the stack represented by a list. |
| E remove(int index) | It is used to remove the element at the specified position in a list. |
| boolean remove(Object o) | It is used to remove the first occurrence of the specified element in a list. |
| E removeFirst() | It removes and returns the first element from a list. |
| boolean removeFirstOccurrence(Object o) | It is used to remove the first occurrence of the specified element in a list (when traversing the list from head to tail). |
| E removeLast() | It removes and returns the last element from a list. |
| boolean removeLastOccurrence(Object o) | It removes the last occurrence of the specified element in a list (when traversing the list from head to tail). |
| E set(int index, E element) | It replaces the element at the specified position in a list with the specified element. |

**Difference between ArrayList and LinkedList**

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes.

|  |  |
| --- | --- |
| **ArrayList** | **LinkedList** |
| 1) ArrayList internally uses a **dynamic array** to store the elements. | LinkedList internally uses a **doubly linked list** to store the elements. |
| 2) Manipulation with ArrayList is **slow** because it internally uses an 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 a doubly linked list, so no bit shifting is required in memory. |
| 3) An 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. |

**List**

**List** in Java provides the facility to maintain the ordered collection. It contains the index-based methods to insert, update, delete and search the elements. It can have the duplicate elements also. We can also store the null elements in the list.

**Java List Methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(int index, E element) | It is used to insert the specified element at the specified position in a list. |
| boolean addAll(int index, Collection<? extends E> c) | It is used to append all the elements in the specified collection, starting at the specified position of the list. |
| E get(int index) | It is used to fetch the element from the particular position of the 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. |
| 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. |
| E remove(int index) | It is used to remove the element present at the specified position in the list. |  |
| void replaceAll(UnaryOperator<E> operator) | It is used to replace all the elements from the list with the specified element. |  |
| E set(int index, E element) | It is used to replace the specified element in the list, present at the specified position. |  |
| void sort(Comparator<? super E> c) | It is used to sort the elements of the list on the basis of specified comparator. |  |
| Spliterator<E> spliterator() | It is used to create spliterator over the elements in a list. |  |
| List<E> subList(int fromIndex, int toIndex) | It is used to fetch all the elements lies within the given range. |  |

 //Converting ArrayList to Array

 String[] array = fruitList.toArray(**new** String[fruitList.size()]);

**Methods of Java ListIterator Interface:**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void add(E e) | This method inserts the specified element into the list. |
| boolean hasNext() | This method returns true if the list iterator has more elements while traversing the list in the forward direction. |
| E next() | This method returns the next element in the list and advances the cursor position. |
| int nextIndex() | This method returns the index of the element that would be returned by a subsequent call to next() |
| boolean hasPrevious() | This method returns true if this list iterator has more elements while traversing the list in the reverse direction. |
| E previous() | This method returns the previous element in the list and moves the cursor position backward. |
| E previousIndex() | This method returns the index of the element that would be returned by a subsequent call to previous(). |
| void remove() | This method removes the last element from the list that was returned by next() or previous() methods |
| void set(E e) | This method replaces the last element returned by next() or previous() methods with the specified element. |

**Java HashSet**

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.

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

HashSet contains unique elements only, allows null value, is non synchronized, doesn't maintain the insertion order.

Here, elements are inserted on the basis of their hashcode.

HashSet is the best approach for search operations.

The initial default capacity of HashSet is 16, and the load factor is 0.75.

Difference between List and Set

A list can contain duplicate elements whereas Set contains unique elements only.

**Methods of Java HashSet class**

Various methods of Java HashSet class are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Method** | **Description** |
| 1) | 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 the set. |

 HashSet<String> set=**new** HashSet();

           set.add("One");

           Iterator<String> i=set.iterator();

**while**(i.hasNext())

           {

           System.out.println(i.next());

           }

 }

**Java LinkedHashSet class**

Java LinkedHashSet class is a Hashtable and Linked list implementation of the set interface.

Contains unique elements only like HashSet; provides all optional set operation and permits null elements

Is non synchronized; maintains insertion order.

LinkedHashSet<String> set=**new** LinkedHashSet();

**Java TreeSet class**

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

Contains unique elements only like HashSet; access and retrieval times are quiet fast ; doesn't allow null element ; is non synchronized; maintains ascending order.

**Methods of Java TreeSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| E ceiling(E e) | It returns the equal or closest greatest element of the specified element from the set, or null there is no such element. |
| Comparator<? super E> comparator() | It returns comparator that arranged elements in order. |
| Iterator descendingIterator() | It is used iterate the elements in descending order. |
| NavigableSet descendingSet() | It returns the elements in reverse order. |
| E floor(E e) | It returns the equal or closest least element of the specified element from the set, or null there is no such element. |
| SortedSet headSet(E toElement) | It returns the group of elements that are less than the specified element. |
| NavigableSet headSet(E toElement, boolean inclusive) | It returns the group of elements that are less than or equal to(if, inclusive is true) the specified element. |
| E higher(E e) | It returns the closest greatest element of the specified element from the set, or null there is no such element. |
| E lower(E e) | It returns the closest least element of the specified element from the set, or null there is no such element. |
| E pollFirst() | It is used to retrieve and remove the lowest(first) element. |
| E pollLast() | It is used to retrieve and remove the highest(last) element. |
| Spliterator spliterator() | It is used to create a late-binding and fail-fast spliterator over the elements. |
| NavigableSet subSet(E fromElement, boolean fromInclusive, E toElement, boolean toInclusive) | It returns a set of elements that lie between the given range. |
| SortedSet subSet(E fromElement, E toElement)) | It returns a set of elements that lie between the given range which includes fromElement and excludes toElement. |
| SortedSet tailSet(E fromElement) | It returns a set of elements that are greater than or equal to the specified element. |
| NavigableSet tailSet(E fromElement, boolean inclusive) | It returns a set of elements that are greater than or equal to (if, inclusive is true) the specified element. |
| Object clone() | It returns a shallow copy of this TreeSet instance. |
| E first() | It returns the first (lowest) element currently in this sorted set. |
| E last() | It returns the last (highest) element currently in this sorted set. |

 TreeSet<String> al=**new** TreeSet<String>();

Java TreeSet Example 3:

Let's see an example to retrieve and remove the highest and lowest Value.

 TreeSet<Integer> set=**new** TreeSet<Integer>();

         set.add(24);

         System.out.println("Highest Value: "+set.pollFirst());

         System.out.println("Lowest Value: "+set.pollLast());

In this example, we perform various NavigableSet operations.

TreeSet<String> set=**new** TreeSet<String>();

         set.add("A");  set.add("B");  set.add("C");  set.add("D");  set.add("E");

         System.out.println("Initial Set: "+set);

         System.out.println("Reverse Set: "+set.descendingSet());

         System.out.println("Head Set: "+set.headSet("C", **true**));

         System.out.println("SubSet: "+set.subSet("A", **false**, "E", **true**));

         System.out.println("TailSet: "+set.tailSet("C", **false**));

 }

}

Output:

Initial Set: [A, B, C, D, E] Reverse Set: [E, D, C, B, A] Head Set: [A, B, C] SubSet: [B, C, D, E] TailSet: [D, E]

**Java Queue Interface**

Java Queue interface orders the element in FIFO(First In First Out) manner. In FIFO, first element is removed first and last element is removed at last.

**Methods of Java Queue Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean offer(object) | It is used to insert the specified element into this queue. |
| Object remove() | It is used to retrieves and removes the head of this queue. |
| Object poll() | It is used to retrieves and removes the head of this queue, or returns null if this queue is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this queue. |
| Object peek() | It is used to retrieves, but does not remove, the head of this queue, or returns null if this queue is empty. |

**PriorityQueue class**

The PriorityQueue class provides the facility of using queue. But it does not orders the elements in FIFO manner.

PriorityQueue<String> queue=**new** PriorityQueue<String>();

queue.add("Amit");

Iterator itr=queue.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

}

**Deque Interface**

Java Deque Interface is a linear collection that supports element insertion and removal at both ends. Deque is an acronym for **"double ended queue".**

**Methods of Java Deque Interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| boolean offer(object) | It is used to insert the specified element into this deque. |
| Object remove() | It is used to retrieves and removes the head of this deque. |
| Object poll() | It is used to retrieves and removes the head of this deque, or returns null if this deque is empty. |
| Object element() | It is used to retrieves, but does not remove, the head of this deque. |
| Object peek() | It is used to retrieves, but does not remove, the head of this deque, or returns null if this deque is empty. |

**ArrayDeque class**

provides the facility of using deque and resizable-array.

Unlike Queue, we can add or remove elements from both sides. Null elements are not allowed; is not thread safe, in the absence of external synchronization. has no capacity restrictions; is faster than LinkedList and Stack.

   Deque<String> deque = **new** ArrayDeque<String>();

   deque.add("Ravi");

**Map Interface**

HashMap and LinkedHashMap allow null keys and values, but TreeMap doesn't allow any null key or value.

A Map can't be traversed, so you need to convert it into Set using keySet() or entrySet() method.

|  |  |
| --- | --- |
| **Class** | **Description** |
| [HashMap](https://www.javatpoint.com/java-hashmap) | HashMap is the implementation of Map, but it doesn't maintain any order. |
| [LinkedHashMap](https://www.javatpoint.com/java-linkedhashmap) | LinkedHashMap is the implementation of Map. It inherits HashMap class. It maintains insertion order. |
| [TreeMap](https://www.javatpoint.com/java-treemap) | TreeMap is the implementation of Map and SortedMap. It maintains ascending order. |

**Useful methods of Map interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| Set keySet() | It returns the Set view containing all the keys. |
| Set<Map.Entry<K,V>> entrySet() | It returns the Set view containing all the keys and values. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection values() | It returns a collection view of the values contained in the map. |

**Methods of Map.Entry interface**

|  |  |
| --- | --- |
| **Method** | **Description** |
| K getKey() | It is used to obtain a key. |
| V getValue() | It is used to obtain value. |
| int hashCode() | It is used to obtain hashCode. |
| V setValue(V value) | It is used to replace the value corresponding to this entry with the specified value. |
| boolean equals(Object o) | It is used to compare the specified object with the other existing objects. |
| static <K extends Comparable<? super K>,V> Comparator<Map.Entry<K,V>> comparingByKey() | It returns a comparator that compare the objects in natural order on key. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByKey(Comparator<? super K> cmp) | It returns a comparator that compare the objects by key using the given Comparator. |
| static <K,V extends Comparable<? super V>> Comparator<Map.Entry<K,V>> comparingByValue() | It returns a comparator that compare the objects in natural order on value. |
| static <K,V> Comparator<Map.Entry<K,V>> comparingByValue(Comparator<? super V> cmp) | It returns a comparator that compare the objects by value using the given Comparator. |

  Map<Integer,String> map=**new** HashMap<Integer,String>();

 map.entrySet().stream().sorted(Map.Entry.comparingByKey()).forEach(System.out::println);

 map.entrySet().stream().sorted(Map.Entry.comparingByKey(Comparator.reverseOrder()))

 .forEach(System.out::println);

**HashMap**

it is not synchronized. It allows us to store the null elements as well, but there should be only one null key.

maintains no order. The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

**Methods of Java HashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object clone() | It is used to return a shallow copy of this HashMap instance: the keys and values themselves are not cloned. |
| Set entrySet() | It is used to return a collection view of the mappings contained in this map. |
| Set keySet() | It is used to return a set view of the keys contained in this map. |
| V put(Object key, Object value) | It is used to insert an entry in the map. |
| void putAll(Map map) | It is used to insert the specified map in the map. |
| V putIfAbsent(K key, V value) | It inserts the specified value with the specified key in the map only if it is not already specified. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the map. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the map, else return false. |
| boolean containsKey(Object key) | This method returns true if some key equal to the key exists within the map, else return false. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a collection view of the values contained in the map. |

  HashMap<Integer,String> map=**new** HashMap<Integer,String>();//Creating HashMap

**Difference between HashSet and HashMap**

HashSet contains only values whereas HashMap contains an entry(key and value).

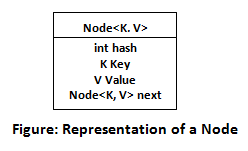
**Working of HashMap in Java**

What is Hashing

It is the process of converting an object into an integer value. The integer value helps in indexing and faster searches.

What is HashMap

HashMap contains an array of the nodes, and the node is represented as a class. It uses an array and LinkedList data structure internally for storing Key and Value. There are four fields in HashMap.



**equals():** It checks the equality of two objects. It compares the Key, whether they are equal or not. It is a method of the Object class. It can be overridden. If you override the equals() method, then it is mandatory to override the hashCode() method.

**hashCode():** This is the method of the object class. It returns the memory reference of the object in integer form. The value received from the method is used as the bucket number. The bucket number is the address of the element inside the map. Hash code of null Key is 0.

**Buckets:** Array of the node is called buckets. Each node has a data structure like a LinkedList. More than one node can share the same bucket. It may be different in capacity.



**Insert Key, Value pair in HashMap**

We use put() method to insert the Key and Value pair in the HashMap. The default size of HashMap is 16 (0 to 15).

Example

In the following example, we want to insert three (Key, Value) pair in the HashMap.

HashMap<String, Integer> map = **new** HashMap<>();

map.put("Aman", 19);

map.put("Sunny", 29);

map.put("Ritesh", 39);

Let's see at which index the Key, value pair will be saved into HashMap. When we call the put() method, then it calculates the hash code of the Key "Aman." Suppose the hash code of "Aman" is 2657860. To store the Key in memory, we have to calculate the index.

Calculating Index

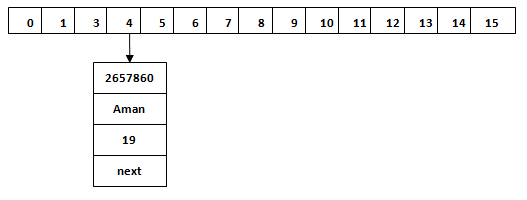
Index minimizes the size of the array. The Formula for calculating the index is:

Index = hashcode(Key) & (n-1)

Where n is the size of the array. Hence the index value for "Aman" is:

Index = 2657860 & (16-1) = 4

The value 4 is the computed index value where the Key and value will store in HashMap.

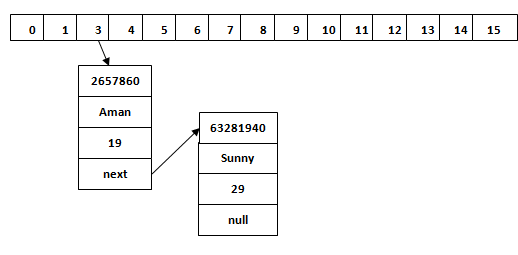


Hash Collision

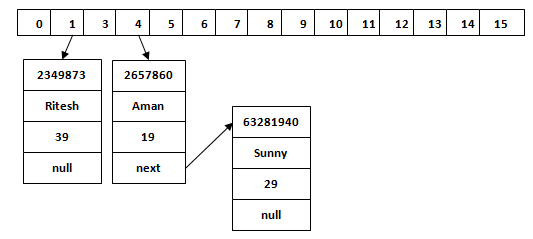
This is the case when the calculated index value is the same for two or more Keys. Let's calculate the hash code for another Key "Sunny." Suppose the hash code for "Sunny" is 63281940. To store the Key in the memory, we have to calculate index by using the index formula.

Index=63281940 & (16-1) = 4

The value 4 is the computed index value where the Key will be stored in HashMap. In this case, equals() method check that both Keys are equal or not. If Keys are same, replace the value with the current value. Otherwise, connect this node object to the existing node object through the LinkedList. Hence both Keys will be stored at index 4.



Similarly, we will store the Key "Ritesh." Suppose hash code for the Key is 2349873. The index value will be 1. Hence this Key will be stored at index 1.



get() method in HashMap

get() method is used to get the value by its Key. It will not fetch the value if you don't know the Key. When get(K Key) method is called, it calculates the hash code of the Key.

Suppose we have to fetch the Key "Aman." The following method will be called.

map.get(**new** Key("Aman"));

It generates the hash code as 2657860. Now calculate the index value of 2657860 by using index formula. The index value will be 4, as we have calculated above. get() method search for the index value 4. It compares the first element Key with the given Key. If both keys are equal, then it returns the value else check for the next element in the node if it exists. In our scenario, it is found as the first element of the node and return the value 19.

Let's fetch another Key "Sunny."

The hash code of the Key "Sunny" is 63281940. The calculated index value of 63281940 is 4, as we have calculated for put() method. Go to index 4 of the array and compare the first element's Key with the given Key. It also compares Keys. In our scenario, the given Key is the second element, and the next of the node is null. It compares the second element Key with the specified Key and returns the value 29. It returns null if the next of the node is null.

**LinkedHashMap**

Is Hashtable and Linked list implementation of the Map interface, with predictable iteration order.

may have one null key and multiple null values; is non synchronized; maintains insertion order.

The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

**Methods of Java LinkedHashMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| V get(Object key) | It returns the value to which the specified key is mapped. |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| Set<Map.Entry<K,V>> entrySet() | It returns a Set view of the mappings contained in the map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped or defaultValue if this map contains no mapping for the key. |
| Set<K> keySet() | It returns a Set view of the keys contained in the map |
| protected boolean removeEldestEntry(Map.Entry<K,V> eldest) | It returns true on removing its eldest entry. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| Collection<V> values() | It returns a Collection view of the values contained in this map. |

 LinkedHashMap<Integer,String> hm=**new** LinkedHashMap<Integer,String>();

**TreeMap**

is a red-black tree based implementation; cannot have a null key but can have multiple null values; is non synchronized; maintains ascending order.

**Methods of Java TreeMap class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Map.Entry<K,V> ceilingEntry(K key) | It returns the key-value pair having the least key, greater than or equal to the specified key, or null if there is no such key. |
| K ceilingKey(K key) | It returns the least key, greater than the specified key or null if there is no such key. |
| Object clone() | It returns a shallow copy of TreeMap instance. |
| Comparator<? super K> comparator() | It returns the comparator that arranges the key in order, or null if the map uses the natural ordering. |
| NavigableSet<K> descendingKeySet() | It returns a reverse order NavigableSet view of the keys contained in the map. |
| NavigableMap<K,V> descendingMap() | It returns the specified key-value pairs in descending order. |
| Map.Entry firstEntry() | It returns the key-value pair having the least key. |
| Map.Entry<K,V> floorEntry(K key) | It returns the greatest key, less than or equal to the specified key, or null if there is no such key. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| SortedMap<K,V> headMap(K toKey) | It returns the key-value pairs whose keys are strictly less than toKey. |
| NavigableMap<K,V> headMap(K toKey, boolean inclusive) | It returns the key-value pairs whose keys are less than (or equal to if inclusive is true) toKey. |
| Map.Entry<K,V> higherEntry(K key) | It returns the least key strictly greater than the given key, or null if there is no such key. |
| K higherKey(K key) | It is used to return true if this map contains a mapping for the specified key. |
| Set keySet() | It returns the collection of keys exist in the map. |
| Map.Entry<K,V> lastEntry() | It returns the key-value pair having the greatest key, or null if there is no such key. |
| Map.Entry<K,V> lowerEntry(K key) | It returns a key-value mapping associated with the greatest key strictly less than the given key, or null if there is no such key. |
| K lowerKey(K key) | It returns the greatest key strictly less than the given key, or null if there is no such key. |
| NavigableSet<K> navigableKeySet() | It returns a NavigableSet view of the keys contained in this map. |
| Map.Entry<K,V> pollFirstEntry() | It removes and returns a key-value mapping associated with the least key in this map, or null if the map is empty. |
| Map.Entry<K,V> pollLastEntry() | It removes and returns a key-value mapping associated with the greatest key in this map, or null if the map is empty. |
| V put(K key, V value) | It inserts the specified value with the specified key in the map. |
| void putAll(Map<? extends K,? extends V> map) | It is used to copy all the key-value pair from one map to another map. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| NavigableMap<K,V> subMap(K fromKey, boolean fromInclusive, K toKey, boolean toInclusive) | It returns key-value pairs whose keys range from fromKey to toKey. |
| SortedMap<K,V> subMap(K fromKey, K toKey) | It returns key-value pairs whose keys range from fromKey, inclusive, to toKey, exclusive. |
| SortedMap<K,V> tailMap(K fromKey) | It returns key-value pairs whose keys are greater than or equal to fromKey. |
| NavigableMap<K,V> tailMap(K fromKey, boolean inclusive) | It returns key-value pairs whose keys are greater than (or equal to, if inclusive is true) fromKey. |
| boolean containsKey(Object key) | It returns true if the map contains a mapping for the specified key. |
| boolean containsValue(Object value) | It returns true if the map maps one or more keys to the specified value. |
| K firstKey() | It is used to return the first (lowest) key currently in this sorted map. |
| V get(Object key) | It is used to return the value to which the map maps the specified key. |
| K lastKey() | It is used to return the last (highest) key currently in the sorted map. |
| Set<Map.Entry<K,V>> entrySet() | It returns a set view of the mappings contained in the map. |
| Collection values() | It returns a collection view of the values contained in the map. |

  TreeMap<Integer,String> map=**new** TreeMap<Integer,String>();

  NavigableMap<Integer,String> map=**new** TreeMap<Integer,String>();

  SortedMap<Integer,String> map=**new** TreeMap<Integer,String>();

**What is difference between HashMap and TreeMap?**

|  |  |
| --- | --- |
| **HashMap** | **TreeMap** |
| 1) HashMap can contain one null key. | TreeMap cannot contain any null key. |
| 2) HashMap maintains no order. | TreeMap maintains ascending order. |

**Hashtable**

Is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.

Contains unique elements; doesn't allow null key or value; is synchronized

The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

**Methods of Java Hashtable class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| Object clone() | It returns a shallow copy of the Hashtable. |
| V compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| V computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction) | It is used to compute its value using the given mapping function, if the specified key is not already associated with a value (or is mapped to null), and enters it into this map unless null. |
| V computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction) | It is used to compute a new mapping given the key and its current mapped value if the value for the specified key is present and non-null. |
| Enumeration elements() | It returns an enumeration of the values in the hash table. |
| Set<Map.Entry<K,V>> entrySet() | It returns a set view of the mappings contained in the map. |
| void forEach(BiConsumer<? super K,? super V> action) | It performs the given action for each entry in the map until all entries have been processed or the action throws an exception. |
| V getOrDefault(Object key, V defaultValue) | It returns the value to which the specified key is mapped, or defaultValue if the map contains no mapping for the key. |
| Enumeration<K> keys() | It returns an enumeration of the keys in the hashtable. |
| Set<K> keySet() | It returns a Set view of the keys contained in the map. |
| V merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction) | If the specified key is not already associated with a value or is associated with null, associates it with the given non-null value. |
| V put(K key, V value) | It inserts the specified value with the specified key in the hash table. |
| void putAll(Map<? extends K,? extends V> t)) | It is used to copy all the key-value pair from map to hashtable. |
| V putIfAbsent(K key, V value) | If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value. |
| boolean remove(Object key, Object value) | It removes the specified values with the associated specified keys from the hashtable. |
| V replace(K key, V value) | It replaces the specified value for a specified key. |
| boolean replace(K key, V oldValue, V newValue) | It replaces the old value with the new value for a specified key. |
| void replaceAll(BiFunction<? super K,? super V,? extends V> function) | It replaces each entry's value with the result of invoking the given function on that entry until all entries have been processed or the function throws an exception. |
| String toString() | It returns a string representation of the Hashtable object. |
| Collection values() | It returns a collection view of the values contained in the map. |
| boolean containsValue(Object value) | This method returns true if some value equal to the value exists within the hash table, else return false. |
| boolean containsKey(Object key) | This method return true if some key equal to the key exists within the hash table, else return false. |
| protected void rehash() | It is used to increase the size of the hash table and rehashes all of its keys. |
| V get(Object key) | This method returns the object that contains the value associated with the key. |

Hashtable<Integer,String> hm=**new** Hashtable<Integer,String>();

 hm.put(100,"Amit");

**Difference between HashMap and Hashtable**

|  |  |
| --- | --- |
| **HashMap** | **Hashtable** |
| 1) HashMap is **non synchronized**. It is not-thread safe and can't be shared between many threads without proper synchronization code. | Hashtable is **synchronized**. It is thread-safe and can be shared with many threads. |
| 2) HashMap **allows one null key and multiple null values**. | Hashtable **doesn't allow any null key or value**. |
| 3) HashMap is a **new class introduced in JDK 1.2**. | Hashtable is a **legacy class**. |
| 4) HashMap is **fast**. | Hashtable is **slow**. |
| 5) We can make the HashMap as synchronized by calling this code Map m = Collections.synchronizedMap(hashMap); | Hashtable is internally synchronized and can't be unsynchronized. |
| 6) HashMap is **traversed by Iterator**. | Hashtable is **traversed by Enumerator and Iterator**. |
| 7) Iterator in HashMap is **fail-fast**. | Enumerator in Hashtable is **not fail-fast**. |
| 8) HashMap inherits **AbstractMap** class. | Hashtable inherits **Dictionary** class. |

**EnumSet**

**Methods of Java EnumSet class**

|  |  |
| --- | --- |
| **Method** | **Description** |
| static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType) | It is used to create an enum set containing all of the elements in the specified element type. |
| static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c) | It is used to create an enum set initialized from the specified collection. |
| static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) | It is used to create an empty enum set with the specified element type. |
| static <E extends Enum<E>> EnumSet<E> of(E e) | It is used to create an enum set initially containing the specified element. |
| static <E extends Enum<E>> EnumSet<E> range(E from, E to) | It is used to create an enum set initially containing the specified elements. |
| EnumSet<E> clone() | It is used to return a copy of this set. |

Java EnumSet Example

**enum** days {

  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

**public** **class** EnumSetExample {

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

    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);

    // Traversing elements

    Iterator<days> iter = set.iterator();

**while** (iter.hasNext())

      System.out.println(iter.next());

  }

**EnumMap**

**Methods of Java EnumMap class**

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 2 | [clone()](https://www.javatpoint.com/post/java-enummap-clone-method) | It is used to copy the mapped value of one map to another map. |
| 3 | [containsKey()](https://www.javatpoint.com/post/java-enummap-containskey-method) | It is used to check whether a specified key is present in this map or not. |
| 4 | [containsValue()](https://www.javatpoint.com/post/java-enummap-containsvalue-method) | It is used to check whether one or more key is associated with a given value or not. |
| 5 | [entrySet()](https://www.javatpoint.com/post/java-enummap-entryset-method) | It is used to create a set of elements contained in the EnumMap. |
| 7 | [get()](https://www.javatpoint.com/post/java-enummap-get-method) | It is used to get the mapped value of the specified key. |
| 9 | [keySet()](https://www.javatpoint.com/post/java-enummap-keyset-method) | It is used to get the set view of the keys contained in the map. |
| 11 | [Values()](https://www.javatpoint.com/post/java-enummap-values-method) | It is used to create a collection view of the values contained in this map. |
| 12 | [put()](https://www.javatpoint.com/post/java-enummap-put-method) | It is used to associate the given value with the given key in this EnumMap. |
| 13 | [putAll()](https://www.javatpoint.com/post/java-enummap-putall-method) | It is used to copy all the mappings from one EnumMap to a new EnumMap. |
| 14 | [remove()](https://www.javatpoint.com/post/java-enummap-remove-method) | It is used to remove the mapping for the given key from EnumMap if the given key is present. |

**Java EnumMap Example**

**import** java.util.\*;

**public** **class** EnumMapExample {

   // create an enum

**public** **enum** Days {

   Monday, Tuesday, Wednesday, Thursday

   };

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

   //create and populate enum map

   EnumMap<Days, String> map = **new** EnumMap<Days, String>(Days.**class**);

   map.put(Days.Monday, "1");

   map.put(Days.Tuesday, "2");

   map.put(Days.Wednesday, "3");

   map.put(Days.Thursday, "4");

   // print the map

**for**(Map.Entry m:map.entrySet()){

       System.out.println(m.getKey()+" "+m.getValue());

      }

   }

}

**Collections**

Java collection class is used exclusively with static methods that operate on or return collections. It inherits Object class.

|  |  |  |  |
| --- | --- | --- | --- |
| **SN** | **Modifier & Type** | **Methods** | **Descriptions** |
| 1) | static <T> boolean | [addAll()](https://www.javatpoint.com/java-collections-addall-method) | It is used to adds all of the specified elements to the specified collection. |
| 2) | static <T> Queue<T> | [asLifoQueue()](https://www.javatpoint.com/java-collections-aslifoqueue-method) | It returns a view of a Deque as a Last-in-first-out (LIFO) Queue. |
| 3) | static <T> int | [binarySearch()](https://www.javatpoint.com/java-collections-binarysearch-method) | It searches the list for the specified object and returns their position in a sorted list. |
| 4) | static <E> Collection<E> | [checkedCollection()](https://www.javatpoint.com/java-collections-checkedcollection-method) | It is used to returns a dynamically typesafe view of the specified collection. |
| 5) | static <E> List<E> | [checkedList()](https://www.javatpoint.com/java-collections-checkedlist-method) | It is used to returns a dynamically typesafe view of the specified list. |
| 6) | static <K,V> Map<K,V> | [checkedMap()](https://www.javatpoint.com/java-collections-checkedmap-method) | It is used to returns a dynamically typesafe view of the specified map. |
| 7) | static <K,V> NavigableMap<K,V> | [checkedNavigableMap()](https://www.javatpoint.com/java-collections-checkednavigablemap-method) | It is used to returns a dynamically typesafe view of the specified navigable map. |
| 8) | static <E> NavigableSet<E> | [checkedNavigableSet()](https://www.javatpoint.com/java-collections-checkednavigableset-method) | It is used to returns a dynamically typesafe view of the specified navigable set. |
| 9) | static <E> Queue<E> | [checkedQueue()](https://www.javatpoint.com/java-collections-checkedqueue-method) | It is used to returns a dynamically typesafe view of the specified queue. |
| 10) | static <E> Set<E> | [checkedSet()](https://www.javatpoint.com/java-collections-checkedset-method) | It is used to returns a dynamically typesafe view of the specified set. |
| 11) | static <K,V> SortedMap<K,V> | [checkedSortedMap()](https://www.javatpoint.com/java-collections-checkedsortedmap-method) | It is used to returns a dynamically typesafe view of the specified sorted map. |
| 12) | static <E> SortedSet<E> | [checkedSortedSet()](https://www.javatpoint.com/java-collections-checkedsortedset-method) | It is used to returns a dynamically typesafe view of the specified sorted set. |
| 13) | static <T> void | [copy()](https://www.javatpoint.com/java-collections-copy-method) | It is used to copy all the elements from one list into another list. |
| 14) | static boolean | [disjoint()](https://www.javatpoint.com/java-collections-disjoint-method) | It returns true if the two specified collections have no elements in common. |
| 15) | static <T> Enumeration<T> | [emptyEnumeration()](https://www.javatpoint.com/java-collections-emptyenumeration-method) | It is used to get an enumeration that has no elements. |
| 16) | static <T> Iterator<T> | [emptyIterator()](https://www.javatpoint.com/java-collections-emptyiterator-method) | It is used to get an Iterator that has no elements. |
| 17) | static <T> List<T> | [emptyList()](https://www.javatpoint.com/java-collections-emptylist-method) | It is used to get a List that has no elements. |
| 18) | static <T> ListIterator<T> | [emptyListIterator()](https://www.javatpoint.com/java-collections-emptylistiterator-method) | It is used to get a List Iterator that has no elements. |
| 19) | static <K,V> Map<K,V> | [emptyMap()](https://www.javatpoint.com/java-collections-emptymap-method) | It returns an empty map which is immutable. |
| 20) | static <K,V> NavigableMap<K,V> | [emptyNavigableMap()](https://www.javatpoint.com/java-collections-emptynavigablemap-method) | It returns an empty navigable map which is immutable. |
| 21) | static <E> NavigableSet<E> | [emptyNavigableSet()](https://www.javatpoint.com/java-collections-emptynavigableset-method) | It is used to get an empty navigable set which is immutable in nature. |
| 22) | static <T> Set<T> | [emptySet()](https://www.javatpoint.com/java-collections-emptyset-method) | It is used to get the set that has no elements. |
| 23) | static <K,V> SortedMap<K,V> | [emptySortedMap()](https://www.javatpoint.com/java-collections-emptysortedmap-method) | It returns an empty sorted map which is immutable. |
| 24) | static <E> SortedSet<E> | [emptySortedSet()](https://www.javatpoint.com/java-collections-emptysortedset-method) | It is used to get the sorted set that has no elements. |
| 25) | static <T> Enumeration<T> | [enumeration()](https://www.javatpoint.com/java-collections-enumeration-method) | It is used to get the enumeration over the specified collection. |
| 26) | static <T> void | [fill()](https://www.javatpoint.com/java-collections-fill-method) | It is used to replace all of the elements of the specified list with the specified elements. |
| 27) | static int | [frequency()](https://www.javatpoint.com/java-collections-frequency-method) | It is used to get the number of elements in the specified collection equal to the specified object. |
| 28) | static int | [indexOfSubList()](https://www.javatpoint.com/java-collections-indexofsublist-method) | It is used to get the starting position of the first occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list. |
| 29) | static int | [lastIndexOfSubList()](https://www.javatpoint.com/java-collections-lastindexofsublist-method) | It is used to get the starting position of the last occurrence of the specified target list within the specified source list. It returns -1 if there is no such occurrence in the specified list. |
| 30) | static <T> ArrayList<T> | [list()](https://www.javatpoint.com/java-collections-list-method) | It is used to get an array list containing the elements returned by the specified enumeration in the order in which they are returned by the enumeration. |
| 31) | static <T extends Object & Comparable<? super T>> T | [max()](https://www.javatpoint.com/java-collections-max-method) | It is used to get the maximum value of the given collection, according to the natural ordering of its elements. |
| 32) | static <T extends Object & Comparable<? super T>> T | [min()](https://www.javatpoint.com/java-collections-min-method) | It is used to get the minimum value of the given collection, according to the natural ordering of its elements. |
| 33) | static <T> List<T> | [nCopies()](https://www.javatpoint.com/java-collections-ncopies-method) | It is used to get an immutable list consisting of **n** copies of the specified object. |
| 34) | static <E> Set<E> | [newSetFromMap()](https://www.javatpoint.com/java-collections-newsetfrommap-method) | It is used to return a set backed by the specified map. |
| 35) | static <T> boolean | [replaceAll()](https://www.javatpoint.com/java-collections-replaceall-method) | It is used to replace all occurrences of one specified value in a list with the other specified value. |
| 36) | static void | [reverse()](https://www.javatpoint.com/java-collections-reverse-method) | It is used to reverse the order of the elements in the specified list. |
| 37) | static <T> Comparator<T> | [reverseOrder()](https://www.javatpoint.com/java-collections-reverseorder-method) | It is used to get the comparator that imposes the reverse of the natural ordering on a collection of objects which implement the Comparable interface. |
| 38) | static void | [rotate()](https://www.javatpoint.com/java-collections-rotate-method) | It is used to rotate the elements in the specified list by a given distance. |
| 39) | static void | [shuffle()](https://www.javatpoint.com/java-collections-shuffle-method) | It is used to randomly reorders the specified list elements using a default randomness. |
| 40) | static <T> Set<T> | [singleton()](https://www.javatpoint.com/java-collections-singleton-method) | It is used to get an immutable set which contains only the specified object. |
| 41) | static <T> List<T> | [singletonList()](https://www.javatpoint.com/java-collections-singletonlist-method) | It is used to get an immutable list which contains only the specified object. |
| 42) | static <K,V> Map<K,V> | [singletonMap()](https://www.javatpoint.com/java-collections-singletonmap-method) | It is used to get an immutable map, mapping only the specified key to the specified value. |
| 43) | static <T extends Comparable<? super T>>void | [sort()](https://www.javatpoint.com/java-collections-sort-method) | It is used to sort the elements presents in the specified list of collection in ascending order. |
| 44) | static void | [swap()](https://www.javatpoint.com/java-collections-swap-method) | It is used to swap the elements at the specified positions in the specified list. |
| 45) | static <T> Collection<T> | [synchronizedCollection()](https://www.javatpoint.com/java-collections-synchronizedcollection-method) | It is used to get a synchronized (thread-safe) collection backed by the specified collection. |
| 46) | static <T> List<T> | [synchronizedList()](https://www.javatpoint.com/java-collections-synchronizedlist-method) | It is used to get a synchronized (thread-safe) collection backed by the specified list. |
| 47) | static <K,V> Map<K,V> | [synchronizedMap()](https://www.javatpoint.com/java-collections-synchronizedmap-method) | It is used to get a synchronized (thread-safe) map backed by the specified map. |
| 48) | static <K,V> NavigableMap<K,V> | [synchronizedNavigableMap()](https://www.javatpoint.com/java-collections-synchronizednavigablemap-method) | It is used to get a synchronized (thread-safe) navigable map backed by the specified navigable map. |
| 49) | static <T> NavigableSet<T> | [synchronizedNavigableSet()](https://www.javatpoint.com/java-collections-synchronizednavigableset-method) | It is used to get a synchronized (thread-safe) navigable set backed by the specified navigable set. |
| 50) | static <T> Set<T> | [synchronizedSet()](https://www.javatpoint.com/java-collections-synchronizedset-method) | It is used to get a synchronized (thread-safe) set backed by the specified set. |
| 51) | static <K,V> SortedMap<K,V> | [synchronizedSortedMap()](https://www.javatpoint.com/java-collections-synchronizedsortedmap-method) | It is used to get a synchronized (thread-safe) sorted map backed by the specified sorted map. |
| 52) | static <T> SortedSet<T> | [synchronizedSortedSet()](https://www.javatpoint.com/java-collections-synchronizedsortedset-method) | It is used to get a synchronized (thread-safe) sorted set backed by the specified sorted set. |
| 53) | static <T> Collection<T> | [unmodifiableCollection()](https://www.javatpoint.com/java-collections-unmodifiablecollection-method) | It is used to get an unmodifiable view of the specified collection. |
| 54) | static <T> List<T> | [unmodifiableList()](https://www.javatpoint.com/java-collections-unmodifiablelist-method) | It is used to get an unmodifiable view of the specified list. |
| 55) | static <K,V> Map<K,V> | [unmodifiableMap()](https://www.javatpoint.com/java-collections-unmodifiablemap-method) | It is used to get an unmodifiable view of the specified map. |
| 56) | static <K,V> NavigableMap<K,V> | [unmodifiableNavigableMap()](https://www.javatpoint.com/java-collections-unmodifiablenavigablemap-method) | It is used to get an unmodifiable view of the specified navigable map. |
| 57) | static <T> NavigableSet<T> | [unmodifiableNavigableSet()](https://www.javatpoint.com/java-collections-unmodifiablenavigableset-method) | It is used to get an unmodifiable view of the specified navigable set. |
| 58) | static <T> Set<T> | [unmodifiableSet()](https://www.javatpoint.com/java-collections-unmodifiableset-method) | It is used to get an unmodifiable view of the specified set. |
| 59) | static <K,V> SortedMap<K,V> | [unmodifiableSortedMap()](https://www.javatpoint.com/java-collections-unmodifiablesortedmap-method) | It is used to get an unmodifiable view of the specified sorted map. |
| 60 | static <T> SortedSet<T> | [unmodifiableSortedSet()](https://www.javatpoint.com/java-collections-unmodifiablesortedset-method) | It is used to get an unmodifiable view of the specified sorted set. |

**Sorting in Collection**

We can sort the elements of:

String objects

Wrapper class objects

User-defined class objects

|  |
| --- |
| **Collections** class provides static methods for sorting the elements of a collection. If collection elements are of a Set type, we can use TreeSet. However, we cannot sort the elements of List. Collections class provides methods for sorting the elements of List type elements. |

**Method of Collections class for sorting List elements**

**public void sort(List list):** is used to sort the elements of List. List elements must be of the Comparable type.

**Note: String class and Wrapper classes implement the Comparable interface. So if you store the objects of string or wrapper classes, it will be Comparable.**

**Example to sort string objects**

ArrayList<String> al=**new** ArrayList<String>();

al.add("Viru");

Collections.sort(al);

Iterator itr=al.iterator();

**while**(itr.hasNext()){

System.out.println(itr.next());

 }

**Example to sort string objects in reverse order**

Collections.sort(al,Collections.reverseOrder());

**Example to sort user-defined class objects**

**class** Student **implements** Comparable<Student> {

**public** String name;

**public** Student(String name) {

**this**.name = name;

  }

**public** **int** compareTo(Student person) {

**return** name.compareTo(person.name);

  }

}

      ArrayList<Student> al=**new** ArrayList<Student>();

      al.add(**new** Student("Viru"));

    Collections.sort(al);

**Java Comparable interface**

Is used to order the objects of the user-defined class. This interface is found in java.lang package and contains only one method named compareTo(Object). It provides a single sorting sequence only, i.e., you can sort the elements on the basis of single data member only. For example, it may be rollno, name, age or anything else.

Java Comparable Example

Let's see the example of the Comparable interface that sorts the list elements on the basis of age.

**class** Student **implements** Comparable<Student>{

**int** age;

**public** **int** compareTo(Student st){

**if**(age==st.age)

**return** 0;

**else** **if**(age>st.age)

**return** 1;

**else**

**return** -1;

}

}

ArrayList<Student> al=**new** ArrayList<Student>();

al.add(**new** Student(101,"Vijay",23));

Collections.sort(al);

**Java Comparator interface**

Is used to order the objects of a user-defined class.

This interface is found in java.util package and contains 2 methods compare(Object obj1,Object obj2) and equals(Object element).

It provides multiple sorting sequences, i.e., you can sort the elements on the basis of any data member, for example, rollno, name, age or anything else.

Methods of Java Comparator Interface

|  |  |
| --- | --- |
| **Method** | **Description** |
| public int compare(Object obj1, Object obj2) | It compares the first object with the second object. |
| public boolean equals(Object obj) | It is used to compare the current object with the specified object. |
| public boolean equals(Object obj) | It is used to compare the current object with the specified object. |

**Java Comparator Example (Non-generic Old Style)**

Let's see the example of sorting the elements of List on the basis of age and name. In this example, we have created 4 java classes:

Student.java AgeComparator.java NameComparator.java Simple.java

**class** Student{  }

**class** AgeComparator **implements** Comparator{

**public** **int** compare(Object o1,Object o2){…..}

**}**

**class** NameComparator **implements** Comparator{

**public** **int** compare(Object o1,Object o2){  …}

**}**

ArrayList al=**new** ArrayList();

al.add(**new** Student(101,"Vijay",23));

Collections.sort(al,**new** NameComparator());

Collections.sort(al,**new** AgeComparator());

**Java Comparator Example (Generic)**

**class** Student{  }

**class** AgeComparator **implements** Comparator<Student>{

**public** **int** compare(Student s1,Student s2){  ….}

}

**class** NameComparator **implements** Comparator<Student>{

**public** **int** compare(Student s1,Student s2){  …}

}

ArrayList<Student> al=**new** ArrayList<Student>();

al.add(**new** Student(101,"Vijay",23));

Collections.sort(al,**new** NameComparator());

Collections.sort(al,**new** AgeComparator());

**Java 8 Comparator interface**

Java 8 Comparator interface is a functional interface that contains only one abstract method. Now, we can use the Comparator interface as the assignment target for a lambda expression or method reference.

**class** Student {    }

  ArrayList<Student> al=**new** ArrayList<Student>();

  al.add(**new** Student(101,"Vijay",23));

/Sorting elements on the basis of name

  Comparator<Student> cm1=Comparator.comparing(Student::getName);

   Collections.sort(al,cm1);

 //Sorting elements on the basis of age

    Comparator<Student> cm2=Comparator.comparing(Student::getAge);

Collections.sort(al,cm2);

Java 8 Comparator Example: nullsFirst() and nullsLast() method

Here, we sort the list of elements that also contains null.

 ArrayList<Student> al=**new** ArrayList<Student>();

 al.add(**new** Student(101,"Vijay",23));

 Comparator<Student> cm1=Comparator.comparing(Student::getName,Comparator.nullsFirst(String::compareTo));

  Collections.sort(al,cm1);

**Properties class**

The **properties** object contains key and value pair both as a string.

It can be used to get property value based on the property key. The Properties class provides methods to get data from the properties file and store data into the properties file. Moreover, it can be used to get the properties of a system.

An Advantage of the properties file

**Recompilation is not required if the information is changed from a properties file:** If any information is changed from the properties file, you don't need to recompile the java class. It is used to store information which is to be changed frequently.

Constructors of Properties class

|  |  |
| --- | --- |
| **Method** | **Description** |
| Properties() | It creates an empty property list with no default values. |
| Properties(Properties defaults) | It creates an empty property list with the specified defaults. |

Methods of Properties class

The commonly used methods of Properties class are given below.

10 Sec

32.6M

633

Difference between JDK, JRE, and JVM

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void load(Reader r) | It loads data from the Reader object. |
| public void load(InputStream is) | It loads data from the InputStream object |
| public void loadFromXML(InputStream in) | It is used to load all of the properties represented by the XML document on the specified input stream into this properties table. |
| public String getProperty(String key) | It returns value based on the key. |
| public String getProperty(String key, String defaultValue) | It searches for the property with the specified key. |
| public void setProperty(String key, String value) | It calls the put method of Hashtable. |
| public void list(PrintStream out) | It is used to print the property list out to the specified output stream. |
| public void list(PrintWriter out)) | It is used to print the property list out to the specified output stream. |
| public Enumeration<?> propertyNames()) | It returns an enumeration of all the keys from the property list. |
| public Set<String> stringPropertyNames() | It returns a set of keys in from property list where the key and its corresponding value are strings. |
| public void store(Writer w, String comment) | It writes the properties in the writer object. |
| public void store(OutputStream os, String comment) | It writes the properties in the OutputStream object. |
| public void storeToXML(OutputStream os, String comment) | It writes the properties in the writer object for generating XML document. |
| public void storeToXML(Writer w, String comment, String encoding) | It writes the properties in the writer object for generating XML document with the specified encoding. |

Example of Properties class to get information from the properties file

To get information from the properties file, create the properties file first.

**db.properties**

user=system

password=oracle

Now, let's create the java class to read the data from the properties file.

**Test.java**

**import** java.util.\*;

**import** java.io.\*;

**public** **class** Test {

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

    FileReader reader=**new** FileReader("db.properties");

    Properties p=**new** Properties();

    p.load(reader);

    System.out.println(p.getProperty("user"));

    System.out.println(p.getProperty("password"));

}

}

oracle

Now if you change the value of the properties file, you don't need to recompile the java class. That means no maintenance problem.

Example of Properties class to get all the system properties

By System.getProperties() method we can get all the properties of the system. Let's create the class that gets information from the system properties.

**Test.java**

**import** java.util.\*;

**import** java.io.\*;

**public** **class** Test {

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

Properties p=System.getProperties();

Set set=p.entrySet();

Iterator itr=set.iterator();

**while**(itr.hasNext()){

Map.Entry entry=(Map.Entry)itr.next();

System.out.println(entry.getKey()+" = "+entry.getValue());

}

}

}

Output:

java.runtime.name = Java(TM) SE Runtime Environment

sun.boot.library.path = C:\Program Files\Java\jdk1.7.0\_01\jre\bin

java.vm.version = 21.1-b02

java.vm.vendor = Oracle Corporation

java.vendor.url = http://java.oracle.com/

path.separator = ;

java.vm.name = Java HotSpot(TM) Client VM

file.encoding.pkg = sun.io

user.country = US

user.script =

sun.java.launcher = SUN\_STANDARD

...........

Example of Properties class to create the properties file

Now let's write the code to create the properties file.

**Test.java**

**import** java.util.\*;

**import** java.io.\*;

**public** **class** Test {

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

Properties p=**new** Properties();

p.setProperty("name","Sonoo Jaiswal");

p.setProperty("email","sonoojaiswal@javatpoint.com");

p.store(**new** FileWriter("info.properties"),"Javatpoint Properties Example");

}

}

Let's see the generated properties file.

**info.properties**

#Javatpoint Properties Example

#Thu Oct 03 22:35:53 IST 2013

email=sonoojaiswal@javatpoint.com

name=Sonoo Jaiswal

**Difference between ArrayList and Vector**

ArrayList and Vector both implements List interface and maintains insertion order.

However, there are many differences between ArrayList and Vector classes that are given below.

|  |  |
| --- | --- |
| **ArrayList** | **Vector** |
| 1) ArrayList is **not synchronized**. | Vector is **synchronized**. |
| 2) ArrayList **increments 50%** of current array size if the number of elements exceeds from its capacity. | Vector **increments 100%** means doubles the array size if the total number of elements exceeds than its capacity. |
| 3) ArrayList is **not a legacy** class. It is introduced in JDK 1.2. | Vector is a **legacy** class. |
| 4) ArrayList is **fast** because it is non-synchronized. | Vector is **slow** because it is synchronized, i.e., in a multithreading environment, it holds the other threads in runnable or non-runnable state until current thread releases the lock of the object. |
| 5) ArrayList uses the **Iterator** interface to traverse the elements. | A Vector can use the **Iterator** interface or **Enumeration** interface to traverse the elements. |

**Java Vector**

**Vector** is like the *dynamic array* which can grow or shrink its size. Unlike array, we can store n-number of elements in it as there is no size limit. It is a part of Java Collection framework since Java 1.2. It is found in the java.util package and implements the *List* interface, so we can use all the methods of List interface here.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

It is similar to the ArrayList, but with two differences-

HTML Tutorial

Vector is synchronized.

Java Vector contains many legacy methods that are not the part of a collections framework.

**Java Vector Methods**

The following are the list of Vector class methods:

|  |  |  |
| --- | --- | --- |
| **SN** | **Method** | **Description** |
| 1) | [add()](https://www.javatpoint.com/java-vector-add-method) | It is used to append the specified element in the given vector. |
| 2) | [addAll()](https://www.javatpoint.com/java-vector-addall-method) | It is used to append all of the elements in the specified collection to the end of this Vector. |
| 3) | [addElement()](https://www.javatpoint.com/java-vector-addelement-method) | It is used to append the specified component to the end of this vector. It increases the vector size by one. |
| 4) | [capacity()](https://www.javatpoint.com/java-vector-capacity-method) | It is used to get the current capacity of this vector. |
| 5) | [clear()](https://www.javatpoint.com/java-vector-clear-method) | It is used to delete all of the elements from this vector. |
| 6) | [clone()](https://www.javatpoint.com/java-vector-clone-method) | It returns a clone of this vector. |
| 7) | [contains()](https://www.javatpoint.com/java-vector-contains-method) | It returns true if the vector contains the specified element. |
| 8) | [containsAll()](https://www.javatpoint.com/java-vector-containsall-method) | It returns true if the vector contains all of the elements in the specified collection. |
| 9) | [copyInto()](https://www.javatpoint.com/java-vector-copyinto-method) | It is used to copy the components of the vector into the specified array. |
| 10) | [elementAt()](https://www.javatpoint.com/java-vector-elementat-method) | It is used to get the component at the specified index. |
| 11) | [elements()](https://www.javatpoint.com/java-vector-elements-method) | It returns an enumeration of the components of a vector. |
| 12) | [ensureCapacity()](https://www.javatpoint.com/java-vector-ensurecapacity-method) | It is used to increase the capacity of the vector which is in use, if necessary. It ensures that the vector can hold at least the number of components specified by the minimum capacity argument. |
| 13) | [equals()](https://www.javatpoint.com/java-vector-equals-method) | It is used to compare the specified object with the vector for equality. |
| 14) | [firstElement()](https://www.javatpoint.com/java-vector-firstelement-method) | It is used to get the first component of the vector. |
| 15) | [forEach()](https://www.javatpoint.com/java-vector-foreach-method) | It is used to perform the given action for each element of the Iterable until all elements have been processed or the action throws an exception. |
| 16) | [get()](https://www.javatpoint.com/java-vector-get-method) | It is used to get an element at the specified position in the vector. |
| 17) | [hashCode()](https://www.javatpoint.com/java-vector-hashcode-method) | It is used to get the hash code value of a vector. |
| 18) | [indexOf()](https://www.javatpoint.com/java-vector-indexof-method) | It is used to get the index of the first occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element. |
| 19) | [insertElementAt()](https://www.javatpoint.com/java-vector-insertelementat-method) | It is used to insert the specified object as a component in the given vector at the specified index. |
| 20) | [isEmpty()](https://www.javatpoint.com/java-vector-isempty-method) | It is used to check if this vector has no components. |
| 21) | [iterator()](https://www.javatpoint.com/java-vector-iterator-method) | It is used to get an iterator over the elements in the list in proper sequence. |
| 22) | [lastElement()](https://www.javatpoint.com/java-vector-lastelement-method) | It is used to get the last component of the vector. |
| 23) | [lastIndexOf()](https://www.javatpoint.com/java-vector-lastindexof-method) | It is used to get the index of the last occurrence of the specified element in the vector. It returns -1 if the vector does not contain the element. |
| 24) | listIterator() | It is used to get a list iterator over the elements in the list in proper sequence. |
| 25) | [remove()](https://www.javatpoint.com/java-vector-remove-method) | It is used to remove the specified element from the vector. If the vector does not contain the element, it is unchanged. |
| 26) | [removeAll()](https://www.javatpoint.com/java-vector-removeall-method) | It is used to delete all the elements from the vector that are present in the specified collection. |
| 27) | [removeAllElements()](https://www.javatpoint.com/java-vector-removeallelements-method) | It is used to remove all elements from the vector and set the size of the vector to zero. |
| 28) | [removeElement()](https://www.javatpoint.com/java-vector-removeelement-method) | It is used to remove the first (lowest-indexed) occurrence of the argument from the vector. |
| 29) | [removeElementAt()](https://www.javatpoint.com/java-vector-removeelementat-method) | It is used to delete the component at the specified index. |
| 30) | removeIf() | It is used to remove all of the elements of the collection that satisfy the given predicate. |
| 31) | removeRange() | It is used to delete all of the elements from the vector whose index is between fromIndex, inclusive and toIndex, exclusive. |
| 32) | [replaceAll()](https://www.javatpoint.com/java-vector-replaceall-method) | It is used to replace each element of the list with the result of applying the operator to that element. |
| 33) | [retainAll()](https://www.javatpoint.com/java-vector-retainall-method) | It is used to retain only that element in the vector which is contained in the specified collection. |
| 34) | set() | It is used to replace the element at the specified position in the vector with the specified element. |
| 35) | setElementAt() | It is used to set the component at the specified index of the vector to the specified object. |
| 36) | setSize() | It is used to set the size of the given vector. |
| 37) | size() | It is used to get the number of components in the given vector. |
| 38) | sort() | It is used to sort the list according to the order induced by the specified Comparator. |
| 39) | spliterator() | It is used to create a late-binding and fail-fast Spliterator over the elements in the list. |
| 40) | subList() | It is used to get a view of the portion of the list between fromIndex, inclusive, and toIndex, exclusive. |
| 41) | toArray() | It is used to get an array containing all of the elements in this vector in correct order. |
| 42) | toString() | It is used to get a string representation of the vector. |
| 43) | trimToSize() | It is used to trim the capacity of the vector to the vector's current size. |

**Java Vector Example**

          Vector<String> vec = **new** Vector<String>();

          vec.add("Tiger");

          vec.addElement("Rat");

**Java Stack**

**Methods of the Stack Class**

We can perform push, pop, peek and search operation on the stack. The Java Stack class provides mainly five methods to perform these operations. Along with this, it also provides all the methods of the [Java Vector class](https://www.javatpoint.com/java-vector).

|  |  |  |
| --- | --- | --- |
| **Method** | **Modifier and Type** | **Method Description** |
| [empty()](https://www.javatpoint.com/java-stack#empty) | boolean | The method checks the stack is empty or not. |
| [push(E item)](https://www.javatpoint.com/java-stack#push) | E | The method pushes (insert) an element onto the top of the stack. |
| [pop()](https://www.javatpoint.com/java-stack#pop) | E | The method removes an element from the top of the stack and returns the same element as the value of that function. |
| [peek()](https://www.javatpoint.com/java-stack#peek) | E | The method looks at the top element of the stack without removing it. |
| [search(Object o)](https://www.javatpoint.com/java-stack#search) | int | The method searches the specified object and returns the position of the object. |

Stack<Integer> stk= **new** Stack<>();

**Java Collection Interface**

Collection is a group of objects, which are known as elements. It is the root interface in the collection hierarchy. This interface is basically used to pass around the collections and manipulate them where the maximum generality is desired.

There are many methods defined in the Collection interface. These are as follows:

|  |  |
| --- | --- |
| **Method** | **Description** |
| [add()](https://www.javatpoint.com/java-collection-add-method) | This method returns a Boolean value true if it inserts the specified element in this collection. |
| [addAll()](https://www.javatpoint.com/java-collection-addall-method) | This method returns a Boolean value true if it adds all the elements of specified collection in the invoking collection. |
| [clear()](https://www.javatpoint.com/java-collection-clear-method) | It removes all the elements automatically from this collection. |
| [contains()](https://www.javatpoint.com/java-collection-contains-method) | It returns a Boolean value true if this queue contains the specified element. |
| [containsAll()](https://www.javatpoint.com/java-collection-containsall-method) | It returns a Boolean value true if this collection contains all the elements in the specified collection. |
| [equals()](https://www.javatpoint.com/java-collection-equals-method) | This method returns a boolean value true if the specified object is equal with this collection. |
| [hashCode()](https://www.javatpoint.com/java-collection-hashcode-method) | It returns a hash code value for this collection. |
| [isEmpty()](https://www.javatpoint.com/java-collection-isempty-method) | This method returns true if this collection contains no elements or is empty. |
| [iterator()](https://www.javatpoint.com/java-collection-iterator-method) | It returns an iterator over the elements in proper sequence. |
| [remove()](https://www.javatpoint.com/java-collection-remove-method) | It removes the specified element from this queue, if it is present in the collection. |
| [removeAll()](https://www.javatpoint.com/java-collection-removeall-method) | It removes all the elements of this collection which are also present in the specified collection. |
| [removeIf()](https://www.javatpoint.com/java-collection-removeif-method) | It removes all the elements of this collection that satisfy the given predicate filter. |
| [retainAll()](https://www.javatpoint.com/java-collection-retainall-method) | This method retains only those elements in this collection that are present in the specified collection. |
| [size()](https://www.javatpoint.com/java-collection-size-method) | It returns the total number of the elements in this collection. |
| [spliterator()](https://www.javatpoint.com/java-collection-spliterator-method) | It returns a spliterator over the elements in this collection. |
| [toArray()](https://www.javatpoint.com/java-collection-toarray-method) | It returns an array containing all the elements of this collection which are in proper sequence. |

**Java ConcurrentHashMap class**

A hash table supporting full concurrency of retrievals and high expected concurrency for updates. This class obeys the same functional specification as Hashtable and includes versions of methods corresponding to each method of Hashtable. However, even though all operations are thread-safe, retrieval operations do not entail locking, and there is not any support for locking the entire table in a way that prevents all access. This class is fully interoperable with Hashtable in programs that rely on its thread safety but not on its synchronization details..

**List of ConcurrentHashMap class Methods**

|  |  |  |
| --- | --- | --- |
| **NO** | **Method** | **Description** |
| 1. | public void [clear()](https://www.javatpoint.com/java-concurrenthashmap-clear-method) | The clear() method of ConcurrentHashMap class removes all of the mappings from this map. |
| 2. | public V [compute(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-compute-method) | The compute() method of ConcurrentHashMap class Attempts to compute a mapping for the specified key and its current mapped value (or null if there is no current mapping). |
| 3. | public V [computeIfAbsent(K key, Function<? super K,? extends V> mappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-computeifabsent-method) | The computeIfAbsent() method of ConcurrentHashMap class attempts to compute its value using the given mapping function and enters it into this map unless null If the specified key is not already associated with a value. |
| 4. | public V [computeIfPresent(K key, BiFunction<? super K,? super V,? extends V> remappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-computeifpresent-method) | The computeIfPresent() method of ConcurrentHashMap class Attempts to compute a new mapping given the key and its current mapped value, If the value for the specified key is present. |
| 5. | public boolean [contains(Object value)](https://www.javatpoint.com/java-concurrenthashmap-contains-method) | The contains() method of ConcurrentHashMap class tests if some key maps into the specified value in this table.. |
| 6. | public boolean [containsKey(Object key)](https://www.javatpoint.com/java-concurrenthashmap-containskey-method) | The containsKey() method of ConcurrentHashMap class tests if the specified object is a key in this table. |
| 7. | public boolean [containsValue(Object value)](https://www.javatpoint.com/java-concurrenthashmap-containsvalue-method) | The containsValue() method of ConcurrentHashMap class returns true if this map maps one or more keys to the specified value. Note: This method may require a full traversal of the map, and is much slower than method containsKey. |
| 8. | public Enumeration<V> [elements()](https://www.javatpoint.com/java-concurrenthashmap-elements-method) | The elements() method of ConcurrentHashMap class returns an enumeration of the values in this table. |
| 9. | public Set<Map.Entry<K,V>> [entrySet()](https://www.javatpoint.com/java-concurrenthashmap-entryset-method) | The entrySet() method of ConcurrentHashMap class Returns a Set view of the mappings contained in this map. The changes made to the map are reflected in the set, and vice-versa. |
| 10. | public boolean [equals(Object o)](https://www.javatpoint.com/java-concurrenthashmap-equals-method) | The elements() method of ConcurrentHashMap class Compares the specified object with this map for equality and returns true if the given object is a map with the same mappings as this map. |
| 11. | public V [get(Object key)](https://www.javatpoint.com/java-concurrenthashmap-get-method) | The get() method of ConcurrentHashMap class Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. |
| 12. | public V [getOrDefault(Object key, V defaultValue)](https://www.javatpoint.com/java-concurrenthashmap-getordefault-method) | The getOrDefault() method of ConcurrentHashMap class Returns the value to which the specified key is mapped, or the given default value if this map contains no mapping for the key. |
| 13. | public int [hashCode()](https://www.javatpoint.com/java-concurrenthashmap-hashcode-method) | The hashcode() method of ConcurrentHashMap class Returns the hash code value for this Map, i.e., the sum of, for each key-value pair in the map, key.hashCode() ^ value.hashCode(). |
| 14. | public Enumeration<K> [keys()](https://www.javatpoint.com/java-concurrenthashmap-keys-method) | The keys() method of ConcurrentHashMap class returns an enumeration of the keys in this table. |
| 15. | public ConcurrentHashMap.KeySetView<K,V> [keySet()](https://www.javatpoint.com/java-concurrenthashmap-keyset-method) public ConcurrentHashMap.KeySetView<K,V> [keySet(V mappedValue)](https://www.javatpoint.com/java-concurrenthashmap-keyset-method) | The keySet() method of ConcurrentHashMap class returns a Set view of the keys contained in this map. The set is stacked by the map, so changes to the map are reflected in the set, and vice-versa. |
| 16. | public long [mappingCount()](https://www.javatpoint.com/java-concurrenthashmap-mappingcount-method) | The mappingCount() method of ConcurrentHashMap class returns the number of mappings. The value returned is an estimated value; the actual count may differ if there are concurrent insertions or removals. |
| 17. | public V [merge(K key, V value, BiFunction<? super V,? super V,? extends V> remappingFunction)](https://www.javatpoint.com/java-concurrenthashmap-merge-method) | The merge() method of ConcurrentHashMap class merge sets If the specified key is not already associated with a (non-null) value, associates it with the given value. |
| 18. | public static <K> ConcurrentHashMap.KeySetView<K,Boolean> [newKeySet()](https://www.javatpoint.com/java-concurrenthashmap-newkeyset-method) public static <K> ConcurrentHashMap.KeySetView<K,Boolean> [newKeySet(int initialCapacity)](https://www.javatpoint.com/java-concurrenthashmap-newkeyset-method) | The newKeySet() method of ConcurrentHashMap class Creates a new Set backed by a ConcurrentHashMap from the given type to Boolean.TRUE. |
| 19. | public V [put(K key, V value)](https://www.javatpoint.com/java-concurrenthashmap-put-method) | The put() method of ConcurrentHashMap class Maps the specified key to the specified value in this table. |
| 20. | public void [putAll(Map<? extends K,? extends V> m)](https://www.javatpoint.com/java-concurrenthashmap-putall-method) | The putAll() method of ConcurrentHashMap class Copies all of the mappings from the specified map to this one. These mappings replace any mappings that this map had for any of the keys currently in the specified map. |
| 21. | public V [putIfAbsent(K key, V value)](https://www.javatpoint.com/java-concurrenthashmap-putifabsent-method) | The putIfAbsent() method of ConcurrentHashMap class Maps If the specified key is not already associated with a value, associates it with the given value. This is equivalent to, for this map. |
| 22. | public V [remove(Object key)](https://www.javatpoint.com/java-concurrenthashmap-remove-method) public boolean [remove(Object key, Object value)](https://www.javatpoint.com/java-concurrenthashmap-remove-method) | The remove() method of ConcurrentHashMap class Removes the key (and its corresponding value) from this map. This method does nothing if the key is not on the map. |
| 23. | public V [replace(K key, V value)](https://www.javatpoint.com/java-concurrenthashmap-replace-method) public boolean [replace(K key, V oldValue, V newValue)](https://www.javatpoint.com/java-concurrenthashmap-replace-method) | The replace() method of ConcurrentHashMap class replaces the entry for a key only if currently mapped to some value. This is equivalent to, for this map. |
| 24. | public String [toString()](https://www.javatpoint.com/java-concurrenthashmap-tostring-method) | The toString() method of ConcurrentHashMap class returns a string representation of this map. The string representation consists of a list of key-value mappings (in no particular order) enclosed in braces ("{}"). |
| 25. | public void [forEach(long parallelismThreshold, BiConsumer<? super K,? super V> action)](https://www.javatpoint.com/java-concurrenthashmap-foreach-method) public <U> void [forEach(long parallelismThreshold, BiFunction<? super K,? super V,? extends U> transformer, Consumer<? super U> action)](https://www.javatpoint.com/java-concurrenthashmap-foreach-method) | The forEach() method of ConcurrentHashMap class Performs the given action for each (key, value). |
| 26. | public Collection<V> [values()](https://www.javatpoint.com/java-concurrenthashmap-values-method) | The values() method of ConcurrentHashMap class returns a Collection view of the values contained in this map. The map backs the collection, so changes to the map are reflected in the collection, and vice-versa. The collection supports element removal, which removes the corresponding mapping from this map, via the Iterator |

 ConcurrentHashMap<String, Integer>  mymap = **new** ConcurrentHashMap<String,  Integer>();

**Java ConcurrentLinkedQueue Class**

ConcurrentLinkedQueue is an unbounded thread-safe queue which arranges the element in FIFO. New elements are added at the tail of this queue and the elements are added from the head of this queue.

ConcurrentLinkedQueue class and its iterator implements all the optional methods of the Queue and Iterator interfaces.

**Methods**

|  |  |
| --- | --- |
| **Methods** | **Description** |
| [add()](https://www.javatpoint.com/java-concurrentlinkedqueue-add-method) | Inserts the specified element at the tail of this queue |
| [addAll()](https://www.javatpoint.com/java-concurrentlinkedqueue-addall-method) | Inserts all the elements which are present in the specified collection to the tail of this queue |
| [contains()](https://www.javatpoint.com/java-concurrentlinkedqueue-contains-method) | Returns true if this queue contains the specified element |
| [forEach()](https://www.javatpoint.com/java-concurrentlinkedqueue-foreach-method) | Performs the given action for each element until all elements have been processed. |
| [isEmpty()](https://www.javatpoint.com/java-concurrentlinkedqueue-isempty-method) | Returns true if this queue contains no elements. |
| [iterator()](https://www.javatpoint.com/java-concurrentlinkedqueue-iterator-method) | Returns an iterator over the elements in this queue |
| [offer()](https://www.javatpoint.com/java-concurrentlinkedqueue-offer-method) | Inserts the specified element at the tail of this queue |
| [remove()](https://www.javatpoint.com/java-concurrentlinkedqueue-remove-method) | Removes the specified element from this queue, if this element is present in the queue |
| [removeAll()](https://www.javatpoint.com/java-concurrentlinkedqueue-removeall-method) | Removes all the elements of this in queue which are present in the specified collection. |
| [removeIf()](https://www.javatpoint.com/java-concurrentlinkedqueue-removeif-method) | Removes all the elements in this queue that satisfy the given predicate filter. |
| [retainAll()](https://www.javatpoint.com/java-concurrentlinkedqueue-retainall-method) | Retain only those elements in this queue that are present in the specified collection. |
| [size()](https://www.javatpoint.com/java-concurrentlinkedqueue-size-method) | Returns the number of the elements in this queue. |
| [spliterator()](https://www.javatpoint.com/java-concurrentlinkedqueue-spliterator-method) | Returns a spliterator over the elements in this queue. |
| [toArray()](https://www.javatpoint.com/java-concurrentlinkedqueue-toarray-method) | Returns an array containing all the elements of this queue which are in proper sequence. |

 ConcurrentLinkedQueue<Integer> queue = **new** ConcurrentLinkedQueue<Integer>();

You can use Java Priority Queue as a Heap.

**Min Heap:** --> to keep the min element always on top, so you can access it in O(1).

PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>();

**Max Heap:** --> to keep the max element always on top, the same order as above.

PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());

Which is the same as (Integer o1, Integer o2) -> Integer.compare(o2, o1) or - Integer.compare(o1, o2) as suggested from other answers.

And you can use:  
add --> to add element to the queue. O(log n)  
remove --> to get and remove the min/max. O(log n)  
peek --> to get, but not remove the min/max. O(1)

Stream In Java

Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)

Last Updated : 09 Oct, 2019

Introduced in Java 8, the Stream API is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.  
The features of Java stream are –

A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.

Streams don’t change the original data structure, they only provide the result as per the pipelined methods.

Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Different Operations On Streams-  
**Intermediate Operations:**

**map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
List number = Arrays.asList(2,3,4,5);  
List square = number.stream().map(x->x\*x).collect(Collectors.toList());

**filter:** The filter method is used to select elements as per the Predicate passed as argument.  
List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());

**sorted:** The sorted method is used to sort the stream.  
List names = Arrays.asList("Reflection","Collection","Stream");  
List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

**collect:** The collect method is used to return the result of the intermediate operations performed on the stream.  
List number = Arrays.asList(2,3,4,5,3);  
Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());

**forEach:** The forEach method is used to iterate through every element of the stream.  
List number = Arrays.asList(2,3,4,5);  
number.stream().map(x->x\*x).forEach(y->System.out.println(y));

**reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
The reduce method takes a BinaryOperator as a parameter.

List number = Arrays.asList(2,3,4,5);  
int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);

Here ans variable is assigned 0 as the initial value and i is added to it .

 List<Float> productPriceList2 =productsList.stream()

                                     .filter(p -> p.price > 30000)// filtering data

                                     .map(p->p.price)        // fetching price

                                     .collect(Collectors.toList()); // collecting as list

        Stream.iterate(1, element->element+1)

        .filter(element->element%5==0)

        .limit(5)

        .forEach(System.out::println);

Output: 5 10 15 20 25