Features of Java

A list of most important features of Java language is given below.

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

### **Simple**

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

* Java syntax is based on C++ (so easier for programmers to learn it after C++).
* Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
* There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

### **Object-oriented**

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts) programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. [Object](https://www.javatpoint.com/object-and-class-in-java)
2. Class
3. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
4. [Polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java)
5. [Abstraction](https://www.javatpoint.com/abstract-class-in-java)
6. [Encapsulation](https://www.javatpoint.com/encapsulation)

### **Platform Independent**

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

### **Robust**

Robust simply means strong. Java is robust because:

* It uses strong memory management.
* There is a lack of pointers that avoids security problems.
* There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
* There are exception handling and the type checking mechanism in Java. All these points make Java robust.

### **Portable**

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

### **Distributed**

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

**Java Virtual machine**

(JVM) is the virtual machine that runs the Java bytecodes. You get this bytecode by compiling the .java files into .class files. .class files contain the bytecodes understood by the JVM.

JVM is responsible to converting Byte code to the machine specific code.

JVM is called *virtual* because it provides a interface that does not depend on the underlying operating system and machine hardware

### *What it does*

*The JVM performs following operation:*

* *Loads code*
* *Verifies code*
* *Memory allocate specific part.*
* *Executes code*
* *Provides runtime environment*

**Memory types:**

* **Method Area** stores class structures like metadata, the constant runtime pool, and the code for methods.

Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

* **Heap** stores all objects that are created during application execution.
* Stacks store local variables, and intermediate results. All such variables are local to the thread by which they are created. Each thread has its own JVM stack, created simultaneously as the thread is created. So all such local variable are called thread-local variables.
* **Program Counter Register**

PC (program counter) register contains the address of the Java virtual machine instruction currently being executed.

This keeps the track of which instruction has been executed and which one is going to be executed. Since instructions are executed by threads, each thread has a separate PC registe

* **Native Method Stack**

It contains all the native methods used in the application.

## JDK vs JRE vs JVM

Let’s look at some of the important difference between JDK, JRE and JVM.

1. **JDK** is for development purpose whereas JRE is for running the java programs.

JDK and JRE both contains JVM so that we can run our java program.

The Java Development Kit (JDK) is a software development environment used for developing Java applications and applets. It includes the Java Runtime Environment (JRE), an interpreter/loader (Java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc) and other tools needed in Java development.

1. **JRE stands for** “Java Runtime Environment” and may also be written as “Java RTE.” The Java Runtime Environment provides the minimum requirements for executing a Java application; it consists of the Java Virtual Machine (JVM), core classes, and supporting files.
2. **(JVM)** is the virtual machine that runs the Java bytecodes. You get this bytecode by compiling the .java files into .class files. .class files contain the bytecodes understood by the JVM.

# **Java Lambda Expressions:**

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

**Functional Interface:**

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

**Why use Lambda Expression:**

1. To provide the implementation of Functional interface.
2. Less coding.

Java lambda expression can be used in the collection framework. It provides efficient and concise way to iterate, filter and fetch data

**Soring Technics:**

Comparable and Comparator both are interfaces and can be used to sort collection elements.

But there are many differences between Comparable and Comparator interfaces that are given below.

* Return 0 if the two objects are the same (i.e., they are “equal,” typically using the equals method of your class)
* Return a negative value if this is less than that
* Return a positive value if this is greater than that

|  |  |
| --- | --- |
| **Comparable** | **Comparator** |
| 1) Comparable provides **single sorting sequence**. In other words, we can sort the collection on the basis of single element such as id or name or price etc. | Comparator provides **multiple sorting sequence**. In other words, we can sort the collection on the basis of multiple elements such as id, name and price etc. |
| 2) Comparable **affects the original class** i.e. actual class is modified. | Comparator **does not affect the original class** i.e. actual class is not modified. |
| 3) Comparable provides **compareTo() method** to sort elements. | Comparator provides **compare() method** to sort elements. |
| 4) Comparable is found in **java.lang** package. | Comparator is found in **java.util** package. |
| 5) We can sort the list elements of Comparable type by **Collections.sort(List)** method. | We can sort the list elements of Comparator type by **Collections.sort(List,Comparator)** method. |

**In scala**

|  |
| --- |
| **class**person(**val**name : *String*, **val**id:Int) { **overridedef**toString = name+" "+id }  **object**DemoScala**extends**App {  **var**a1 = **new** person("nagenda",102)  **var**a2 = **new** person("abc",101)  **var**a3 = **new** person("cdf",10)  **var**a4 = **new** person("def",1)  **var**p = List(a1,a2,a3,a4)  **var**res = p.sortWith(\_.id< \_.id).foreach(println)  **var**str = List("banana", "pear", "apple", "orange")  str.sortWith(sortedByCus).foreach(println)  **def**sortedByCus(a : *String*, b:*String*) : Boolean = a.length<b.length()  }  **In Java:**  **class** Student {  **int**rollno;  String name;  **int**age;  @Override  **public** String toString() {  **return**"Student [rollno=" + rollno + ", name=" + name + ", age=" + age + "]";  }  Student(**int**rollno, String name, **int**age) {  **this**.rollno = rollno;  **this**.name = name;  **this**.age = age;  }  }  **class**nameComparator**implements** Comparator<Student>{  @Override  **publicint**compare(Student o1, Student o2) {  **return**o1.name.compareTo(o2.name);  }  }  **publicclass**DemoJava {  **publicstaticvoid**main(String[] args) {  ArrayList<Student>al=**new**ArrayList<Student>();  al.add(**new** Student(101,"Vijay",23));  al.add(**new** Student(106,"Ajay",10));  al.add(**new** Student(105,"Jai",5));    System.***out***.println("Sorting by Name...");  //Collections.sort(al, new nameComparator());  Collections.*sort*(al,(s1,s2)-> {  //return s1.name.compareTo(s2.name);  **if**(s1.age>s2.age) {  **return** 1;  }**elseif**(s1.age<s2.age) {  **return** -1;  }**else** {  **return** 0;  }    });  System.***out***.println("after by Name...");  al.forEach(i ->System.***out***.println(i));  }  } |

# **Java Functional Interfaces:**

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

**The primary purpose served by Functional Interfaces:**

One of the most important uses of Functional Interfaces is that implementations of their abstract method can be passed around as lambda expressions. By virtue of their ability to pass around functionality(i.e. behavior), functional interfaces primarily enable behavior parameterization.

|  |
| --- |
| @FunctionalInterface  **interface**DemoFunctional**extends**DemoFun {  **publicvoid**show();  **publicdefaultint**doSome(**int**a, **int**b) { **return**a + b; }  **publicdefaultint** doSome1(**int**a, **int**b) { **return**a - b; }  **publicstatic** String doSome2(String str) { **return**str.toUpperCase(); }  **publicint**hashCode();  String toString();  }  **class** Demo **implements**DemoFunctional {  @Override  **publicvoid**show() {  System.***out***.println("Hello");  }  }  **publicclass**FunctionalDemo {  **publicstaticvoid**main(String[] args) {  Demo d = **new**Demo();  d.show();  System.***out***.println(d.doSome(10, 20));  System.***out***.println(d.doSome1(10, 20));  }  } |

**Java StringJoiner:**

Java added a new final class StringJoiner in java.util package. It is used to construct a sequence of characters separated by a delimiter. Now, you can create string by passing delimiters like comma(,), hyphen(-) etc. You can also pass prefix and suffix to the char sequence.

StringJoinerstr1 = **new**StringJoiner(",","[","]");

str1.add("java");

str1.add("scala");

str1.add("spark");

str1.add("kafka");

str1.add("micorService");

System.***out***.println(str1);

outPut: [java,scala,spark,kafka,micorService]

**Default method:**

if we want to add additional methods in the interfaces, it will require change in all the implementing classes. As interface grows old

For creating a default method in java interface, we need to use “default” keyword with the method signature

public interface Interface1 {

void method1(String str);

default void log(String str){

System.out.println("I1 logging::"+str);

} }

when a class will implement Interface1, it is not mandatory to provide implementation for default methods of interface. This feature will help us in extending interfaces with additional methods

1. Java interface default methods will help us in extending interfaces without having the fear of breaking implementation classes.

2. Java 8 interface default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.

3. Java interface default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can choose which one to override.

4. Java interface default methods are also referred to as Defender Methods or Virtual extension methods.

5. if we have Object class methods defined as default methods in interfaces, it will be useless because Object class method will always be used. That’s why to avoid confusion

Default methods enable to add new functionality to existing Interfaces without breaking older implementation of these Interfaces.

When we extend an interface that contains a default method, we can perform the following action:

Not override the default method and will inherit the default method.

Override the default method similar to other methods we have overridden subclasses.

Redeclare default method as abstract, which will force subclasses to override it.

java 8 default methods we can add methods to interface without disturbing existing functionality.So instead of overriding now we can inherit these default methods from interfaces

**Default methods will help us to avoid utility classes.**

We can define utility methods inside the interface and use it in all classes which is implementing.

One of the major reason to introduce this default method in java 8 is to support lambda expressions in collections API and to enhance.

interface A {

default void defMethod(){ System.out.println("Default method of interface: A"); }

**Interface Static Methods in Java 8:**

java 8 interface method is static method.

Now we can define static methods inside interface but we cannot override these static methods.

These static methods will act as helper methods.These methods are the parts of interface not belongs to implementation class objects.

If we want to call these static methods, which are defined, in interfaces, we need to use interface name so that we can access these methods.

interface StaticInterface{

Static void print(String str){

System.out.println("Static method of interface:"+str);

}

}

**Static Methods b/w Default method:**

<access modifier> static <return-parameter> method-name([<method-params>])[throws <Exceptions>]

<access modifier> default <return-parameter> method-name([<method-params>])[throws <Exceptions>]

public default void defaultMethod()

public static void staticMethod()

**Scope for method invocation:**

**When to design\use Static Methods:**

A static method is visible/usable in Interface Scope. Once the interface has been compiled, then the static method can be invoked as –

<Interface-Name>. <static-method-name>([params])

For Example: For the code shown in difference-1 above,

staticMethod() would be invoked as – InterfaceWithDefaultStatic.staticMethod()

A default method is visible\usable in the object instance scope. A Class needs to implement the interface containing the default method, then the default method can be invoked on the instance of the implementing class. The format of invocation of a default method is –

<ObjectInstance-name>.<default-method-name>([params])

**Purpose served in the overall design:**

Static Methods are the utility(util) methods which are associated to an Interface. So, in case you need any util methods which can operate on an Interface’s implementing class’s instances, then add that as a static method to the Interface itself.

whenever you need a static utility method for interface’s implementors, add that static utility method in Interface itself as a static method

**When to design\use Default Methods:**

Default methods we can add methods to interface without disturbing existing functionality.

**Differences between static and default methods**

Default methods **can be** overriden in implementing class, while static **cannot**.

Static method belongs **only** to Interface class, so you can only invoke static method on Interface class, not on class implementing this Interface

**Interface default methods:**

It helps in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.

It helps in extending interfaces without having the fear of breaking implementation classes.

**Interface static methods:**

They are part of interface we cannot use it for implementation class objects.

It helps in providing security by not allowing implementation classes to override them.

**Java Stream:**

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a java Stream is, a data structure that is computed on-demand.

Java Stream doesn’t store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations. Such as we can create a stream from the list and filter it based on a condition.

Java, java.util.Stream represents a stream on which one or more operations can be performed. Stream operations are either intermediate or terminal.

While terminal operations return a result of a certain type,

Intermediate operations return the stream itself so you can chain multiple method calls in a row. Streams are created on a source, e.g. a java.util.Collection like lists or sets (maps are not supported). Stream operations can be executed either sequential or parallel.

Java Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expression. Using lambda expressions make our code readable and short.

All the Java Stream API interfaces and classes are in the java.util.stream package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for primitive types – Instream, LongStream and DoubleStream.

You can consider Collection as in-memory data structure, where you can add,and remove element. Where as in Stream you can perform two kind of operation:

However, if you notice, with stream you cannot add or remove elements.

Stream is kind of iterator, you can traverse collection through stream. Note, you can traverse stream only once.

**Intermediate:**

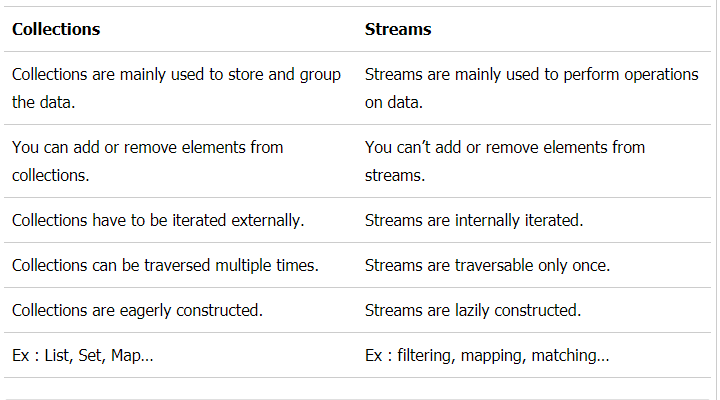
Java Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations. Commonly used intermediate operations are filter and map.

Example operation: Filter, map, sorted, skip, distinct, limit on the result set.

**Terminal:**

Java 8 Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can’t use stream. Terminal operations are eager in nature i.e. they process all the elements in the stream before returning the result. Commonly used terminal methods are forEach, toArray, min, max, findFirst, anyMatch, allMatch etc. You can identify terminal methods from the return type; they will never return a Stream

Example operation: ForEach, reduces, collect, count, min, max andcollect the result set to a collection.



**Function Descriptor:**

In Java 8 a Function Descriptor is a term used to describe the signature of the abstract method of a Functional Interface.

The signature of the abstract method of a Functional Interface is syntactically the same as the signature of the Lambda Expression.

Hence, a Function Descriptor also describes the signature.

|  |
| --- |
| package com.javabrahman.java8;  @FunctionalInterface  public interface FirstInterface {    //Single abstract method    public void singleMethod(String param);  }  the signature of the abstract method OR the function descriptor is (String) -> void |

**Consumer Interface:**

It is a functional interface defined in java.util.function package. It contains an abstract accept () and a default andThen() method. It can be used as the assignment target for a lambda expression or method reference.

The Consumer Interface accepts a single argument and does not return any result.

**BiConsumer Interface:**

BiConsumer Interface accepts two input arguments and does not return any result. This is the two-arity specialization of Consumer interface. It provides a functional method accept(Object, Object) to perform custom operations.

**Function Interface:**

When an object of a type is taken as input and it is converted(or mapped) to another type. Common usage of Function is in streams where in the map function of a stream accepts an instance of Function to convert the stream of one type to a stream of another type.

**Supplier:**

Supplier’s Function Descriptor is () ->T . This means that there is no input in the lambda definition and the return output is an object of type

In all scenarios where there is no input to an operation and it is expected to return an output the in-built functional interface Supplier<T> can be used without the need to define a new functional interface every time.

Supplier<String>helloStrSupplier = () -> new String("Hello");

String helloStr = helloStrSupplier.get();

**BinaryOperator:**

java.util.function.BinaryOperator is a functional interface, which extends the java.util.function.BiFunctioninterface . The BinaryOperator interface represents an operation that takes two arguments of same type and returns a result of same type as its input arguments.

**Note**– The BinaryOperator interface inherits the functional method apply() and default method andThen() from  the BiFunction interface.

|  |
| --- |
| Example 1:  BinaryOperator<Integer> operator1 = (a, b) -> a + b; System.out.println(operator1.apply(5, 7));  BinaryOperator<String> operator2 = (s1, s2) -> s1 + s2; System.out.println(operator2.apply("Hello ", "[BORAJI.COM](http://boraji.com/)”));  O/p:  12 Hello [BORAJI.COM](http://boraji.com/)  Example 2:  BinaryOperator<Integer> operator = (a, b) -> a + b;  Function<Integer, Integer> function = n -> n \* 2;  // Using andThen() System.out.println(operator.andThen(function).apply(1, 6));  o/p:  14  Example 3:  Comparator<Integer> comparator = (a, b) -> (a.compareTo(b));  // Using maxBy()  BinaryOperator<Integer>opMax = BinaryOperator.maxBy(comparator);  System.out.println("Max: " + opMax.apply(5, 6));  System.out.println("Max: " + opMax.apply(9, 6));  // Using minBy()  BinaryOperator<Integer>opMin = BinaryOperator.minBy(comparator);  System.out.println("Min: " + opMin.apply(5, 6));  System.out.println("Min: " + opMin.apply(9, 6));  **o/p**  Max: 6 Max: 9 Min: 5 Min: 6 |

**BiFunction:**

java.util.function.BiFunction is a functional interface whose functional method is R apply(T t, U u). The BiFunction is interface represents an operation that takes two arguments (*T*and *U*) and returns a result *R*.

|  |
| --- |
| BiFunction<String, String, String> function1 = (s1, s2) ->{ String s3 = s1 + s2; return s3; }; System.out.println(function1.apply("BORAJI", ".COM"));  BiFunction<Integer, Integer, Integer> function2 = (a, b) -> a + b; System.out.println(function2.apply(100, 200));  O/P:  BORAJI.COM 300 |

**Examples Streams:**

|  |
| --- |
| Stream.*of*("d2", "a2", "b1", "b3", "c")  .filter(s -> {  System.***out***.println("filter: " + s);  **Return**s.startsWith("a");  })  .sorted((s1, s2) -> {  System.***out***.printf("sort: %s; %s\n", s1, s2);  **return**s1.compareTo(s2);  })  .map(s -> {  System.***out***.println("map: " + s);  **return**s.toUpperCase();  })  .forEach(s ->System.***out***.println("forEach: " + s));  // filter: d2  // filter: a2  // filter: b1  // filter: b3  // filter: c  // map: a2  // forEach: A2 |

Java 8 streams cannot be reused. As soon as you call any terminal operation the stream is closed:

To overcome this limitation we have to to create a new stream chain for every terminal operation we want to execute, e.g. we could create a stream supplier to construct a new stream with all intermediate operations already set up:

|  |
| --- |
| Stream<String>stream =  Stream.*of*("d2", "a2", "b1", "b3", "c")  .filter(s ->s.startsWith("a"));  stream.anyMatch(s ->**true**); // ok  stream.noneMatch(s ->**true**); // exception  //Calling noneMatch after anyMatch on the same stream results in the following exception:  //java.lang.IllegalStateException: stream has already been operated upon or closed    Supplier<Stream<String>>stream1 = () ->  Stream.*of*("d2", "a2", "b1", "b3", "c")  .filter(s ->s.startsWith("a"));  stream1.get().anyMatch(s ->**true**); // ok  stream1.get().noneMatch(s ->**true**); // ok  stream1.get().noneMatch(s ->**true**); // ok  Each call to get() constructs a new stream on which we are save to call the desired terminal operation.  -------------------------------------------------------------------------------------  List<Person>persons = Arrays.*asList*(  **new**Person("mkyong", 30),  **new**Person("jack", 20),  **new**Person("lawrence", 40)  );  Person result1 = persons.stream() // Convert to steam  .filter(x ->"ab".equals(x.getName())) // we want "jack" only  .findAny() // If 'findAny' then return found  .orElse(**null**); // If not found, return nullSystem.***out***.println(result1);  **Map:**  List<String>name = persons.stream()  //.filter(x -> "jack".equals(x.getName()))  .sorted((x1,x2) ->x1.getName().compareTo(x2.getName()))  .map(Person::getName)  .collect(Collectors.*toList*()); //convert stream to String  /\* .findAny()  .orElse("");\*/]  System.***out***.println("name : " + name);  **Stream types:**  Stream<Integer> list = persons.stream().map( p ->p.getAge() \* 2);  list.forEach(System.out::println);    IntStream list1 = persons.stream().mapToInt(p ->p.getAge() \* 2);  list1.forEach(System.out::println);    DoubleStream list2 = persons.stream().mapToDouble(p ->p.getAge() \* 2);  list2.forEach(System.out::println); |

**Collect:**

Collect is an extremely useful terminal operation to transform the elements of the stream into a different kind of result, e.g. a List, Set or Map

|  |
| --- |
| Exampple 1:  List<Person>list= Arrays.*asList*(  **new**Person("Max", 18),  **new**Person("Peter", 23),  **new**Person("Pamela", 23),  **new**Person("David", 12));  List<Person>listPerson = list.stream().filter(p ->p.name.startsWith("P"))  .collect(Collectors.*toList*());  Set<Person>setPerson = list.stream().filter(p ->p.name.startsWith("P"))  .collect(Collectors.*toSet*());  Map<Integer,String>m1 = list.stream().filter(p ->p.name.startsWith("M"))  .collect(Collectors.*toMap*(Person::getAge, Person::getName));    /\* or Map<Integer,String> m6 = list.stream().filter(p ->p.name.startsWith("M"))  .collect(Collectors.toMap( p ->p.age, p -> p.name));\*/    Map<Integer,List<Person>>m2 = list.stream()  .collect(Collectors.*groupingBy*(p ->p.getAge()));    Double m3 = list.stream()  .collect(Collectors.*averagingInt*(p ->p.getAge()));    IntSummaryStatisticsm4 = list.stream()  .collect(Collectors.*summarizingInt*(*p ->p.getAge()));*    String m5 = list.stream().filter(p->p.age>= 18).map(p ->p.name)  .collect(Collectors.*joining*(",", "prefix "," suffext"));    // The join collector accepts a delimiter as well as an optional prefix and suffix.    System.***out***.println("Joing: "+m5);  System.***out***.println("summary statics: "+m4);  m2.forEach((age,p)->System.***out***.format("age %s: %s\n",age,p));  System.***out***.println("Group By Output: " +m2);  System.***out***.println("Map Output: " +m1);  listPerson.forEach(System.***out***::println);  setPerson.forEach(System.***out***::println);  Example 2:  List<Integer>list = Arrays.*asList*(10,2,25,50,36,80,71,20,10);    Optional<Integer>opt1 = list.stream().filter( e ->e>10).collect(Collectors.*maxBy*(Comparator.*comparing*(e ->e)));  System.***out***.println("list integer: "+opt1);    Optional<Integer>opt2 = list.stream().filter( e ->e> 10).collect(Collectors.*minBy*(Comparator.*comparing*(e ->e)));  System.***out***.println("list integer: "+opt2);      Double totalSalaryExpense = *employeeList*.stream().map(emp ->emp.getSalary()).reduce(0.00, (a, b) ->a + b);  System.***out***.println("Total salary expense: " + totalSalaryExpense);    Optional<Employee>maxSalaryEmp1 = *employeeList*.stream().  collect(Collectors.*maxBy*(Comparator.*comparing*(Employee::getSalary)));    Optional<Employee>minSalaryEmp2 = *employeeList*.stream().  collect(Collectors.*minBy*(Comparator.*comparing*((e) ->e.getSalary())));    System.***out***.println("maxmim salary: "+ maxSalaryEmp1);  System.***out***.println("minimum salary: "+ minSalaryEmp2);    Optional<Employee>maxSalaryEmp = *employeeList*.stream().reduce((e1,e2) ->e1.getSalary() >e2.getSalary() ? e1 : e2);  **if**(maxSalaryEmp.isPresent())  System.***out***.println("Employee with max salary: "+maxSalaryEmp.get());  } |

**Parallel Streams:**

Streams can be executed in parallel to increase runtime performance on large amount of input elements

|  |
| --- |
| List<Person> persons = Arrays.asList(  new Person("Max", 18),  new Person("Peter", 23),  new Person("Pamela", 23),  new Person("David", 12));  persons  .parallelStream()  .reduce(0,  (sum, p) -> {  System.out.format("accumulator: sum=%s; person=%s [%s]\n",  sum, p, Thread.currentThread().getName());  return sum += p.age;  },  (sum1, sum2) -> {  System.out.format("combiner: sum1=%s; sum2=%s [%s]\n",  sum1, sum2, Thread.currentThread().getName());  return sum1 + sum2;  }); |

**FlatMap**

Stream.flatMap() helps in converting Collection<Collection<T>> to Collection<T>.

Ex: List<List<String>> into List<Strings>

flatMap() = map() + Flattening

|  |
| --- |
| /\*Stream<List<Integer>> integerListStream = Stream.of(  Arrays.asList(1, 2),  Arrays.asList(3, 4),  Arrays.asList(5)  );  Stream<Integer> integerStream = integerListStream .flatMap(e->e.stream());  integerStream.forEach(System.out::println);\*/    List<List<Integer>> list = **new** ArrayList<>();  list.add(Arrays.*asList*(1, 2));  list.add(Arrays.*asList*(3, 4));  list.add(Arrays.*asList*(5));  list.stream().flatMap(e -> e.stream()).forEach(System.***out***::println); |

**What is flattening**

Flattening is referred by converting several lists of lists, and merge all those lists to create single list containing all the elements from all the lists.

|  |
| --- |
| **publicclass** Student {  **private** String name;  **private** Set<String>book;  **publicvoid**addBook(String book) {  **if** (**this**.book == **null**) {  **this**.book = **new** HashSet<>();  }  **this**.book.add(book);  }  **public** String getName() {  **return**name;  }  **publicvoid**setName(String name) {  **this**.name = name;  }  **public** Set<String>getBook() {  **return**book;  }  **publicvoid**setBook(Set<String>book) {  **this**.book = book;  }  }  Studentobj1 = **new**Student();  obj1.setName("mkyong");  obj1.addBook("Java 8 in Action");  obj1.addBook("Spring Boot in Action");  obj1.addBook("Effective Java (2nd Edition)");  Studentobj2 = **new**Student();  obj2.setName("zilap");  obj2.addBook("Learning Python, 5th Edition");  obj2.addBook("Effective Java (2nd Edition)");  List<Student>list = **new**ArrayList<>();  list.add(obj1);  list.add(obj2);  /\* List<String>listresult = list.stream().map( e ->e.getBook())  .flatMap(e ->e.stream())  .filter(e ->e.startsWith("Spring"))  .collect(Collectors.toList());\*/  Optional<String>opnResults= list.stream().map( e ->e.getBook())  .flatMap(e ->e.stream())  .filter(e ->e.startsWith("Spring"))  .findAny();  opnResults.ifPresent(System.***out***::println); |

**Optional:**

Java 8 has introduced a new class Optional in java.util package. It is used to represent a value is present or absent. The main advantage of this new construct is that No more too many null checks and NullPointerException. It avoids any runtime NullPointerExceptions and supports us in developing clean and neat Java APIs or Applications. Like Collections and arrays, it is also a Container to hold at most one value. Let us explore this new construct with some useful examples.

Advantages of Java 8 Optional:

1. Null checks are not required.
2. No more NullPointerException at run-time.
3. We can develop clean and neat APIs.
4. No more Boiler plate code

|  |
| --- |
| List<String>list = Arrays.*asList*("ab", "db", "c");  Optional<String>opt = list.stream()  .filter(e ->e.startsWith("ab"))  .findAny();  **if** (opt.isPresent()) {  System.***out***.println(opt.get()); // if value present get the value from optional  } **else** {  System.***out***.println("no value found");  }  opt.ifPresent(e ->System.***out***.println(e)); // if value is not found, ifPresent statement executed  List<String>list2 = Arrays.*asList*("ab", "db", "c");  String str = list2.stream().  filter(e ->e.startsWith("dd"))  .findAny()  .orElse("no value found");    System.***out***.println(str);  Optional<String>nonEmptyGender = Optional.of("male");  Optional<String>emptyGender = Optional.empty(); |

**AllMatch,anyMatch andnonmatch:**

**anyMatch() method**. This method returns true if any elements of the Stream matches the given predicate

The **stream noneMatch() method**works just opposite to the anyMatch() method, it returns true if none of the stream elements match the given predicate, it returns false if any of the stream elements matches the condition specified by the predicate.

**allMatch() method**, which returns true if all the elements of stream satisfy the given predicate, else it returns false.

## Streams filter(), findAny() and orElse()

## stream.filter() filter to a List, and .findAny().orElse (null) to return an object conditional.

Person result2 =persons.stream()

.filter(x ->"ahmook".equals(x.getName()))

.findAny()

.orElse(null);

<https://beginnersbook.com/2017/11/java-8-stream-allmatch-example/>

**Exception Handling:**

List<Integer>integers = Arrays.*asList*(3, 9, 7, 0, 10, 20);

integers.forEach(i -> {

**try** {

System.out.println(50 / i);

} **catch** (ArithmeticException e) {

System.err.println(

"Arithmetic Exception occured : " + e.getMessage());

}

});

**static** Consumer<Integer>lambadaWrapper(Consumer<Integer>consumer) {

**return**i -> {

**try**{

consumer.accept(i);

}**catch** (Exception e) {

System.***out***.println("catch Exceptin: "+e);

}

};

}

}

**A Collector is a mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed.**

**Abstract class in Java:**

A class that is declared with abstract keyword, is known as abstract class in java. It can have abstract and non-abstract methods (method with body).

Before learning java abstract class, let's understand the abstraction in java first.

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

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

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

**Ways to achieve Abstraction:**

There are two ways to achieve abstraction in java

Abstract class (0 to 100%)

Interface (100%)

**Abstract class in Java**

A class that is declared as abstract is known as abstract class. It needs to be extended and its method implemented. It cannot be instantiated.

**Abstract method**

|  |
| --- |
| A method that is declared as abstract and does not have implementation is known as abstract method. |

# **Interface in Java:**

An interface in java is a blueprint of a class. It has static constants and abstract methods.The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

In other words, you can say that interfaces can have methods and variables but the methods declared in interface contain only method signature, not body.

Java Interface also represents IS-A relationship.

It cannot be instantiated just like abstract class.

## Why use Java interface?

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

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

## How to declare interface?

Interface is declared by using interface keyword. It provides total abstraction; means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default. A class that implement interface must implement all the methods declared in the interface.

Since Java 8, interface can have default and static methods which is discussed later.

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

As we have explained in the inheritance chapter, multiple inheritance is not supported in case of class because of ambiguity. But it is supported in case of interface because there is no ambiguity as implementation is provided by the implementation class.

**Difference between abstract class and interface:**

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

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

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

**Hashing:**

Hashing is a way to assign a unique code for any variable/object after applying any function/algorithm on its properties.

A true Hashing function must follow this rule:

Hash function should return the same hash code each and every time, when function is applied on same or equal objects. In other words, two equal objects must produce same hash code consistently.

**Hashing**: Hashing is transforming a given entity object to some shorted fixed length value, the shorter value helps in indexing and faster value.

**HashMap Work Internally In Java:**

Whenever hash map is created, in internally array of table will be created by default size is 16 (indexes or buckets each bucket is a node or linked list)

HashMap stores the data in the form of key-value pairs. Each key-value pair is stored in an object of Node<K, V> class, (Entry<K, V> class). class is the static inner class of Hash Map which is defined like below.

static class Entry<K,V> implements Map.Entry<K,V>

{

final K key;

V value;

Entry<K,V> next;

int hash;

//Some methods are defined here

}

|  |
| --- |
| **publicclass** HashMap<K,V>**extends**AbstractMap<K,V>  **implements** Map<K,V>, Cloneable, Serializable {  **privatestaticfinallong*serialVersionUID*** = 362498820763181265L;  /\*\*  \* Basic hash bin node, used for most entries. (See below for  \* TreeNode subclass, and in LinkedHashMap for its Entry subclass.)  \*/  **staticclass**Node<K,V>**implements**Map.Entry<K,V> {  **finalint**hash;  **final** K key;  V value;  Node<K,V>next;  Node(**int**hash, K key, V value, Node<K,V>next) {  **this**.hash = hash;  **this**.key = key;  **this**.value = value;  **this**.next = next;  }  **publicfinal** K getKey() { **return**key; }  **publicfinal** V getValue() { **return**value; }  **publicfinal** String toString() { **return**key + "=" + value; }  **publicfinalint**hashCode() {  **return**Objects.*hashCode*(key) ^ Objects.*hashCode*(value);  }  **publicfinal** V setValue(V newValue) {  V oldValue = value;  value = newValue;  **return**oldValue;  }  **publicfinalboolean**equals(Object o) {  **if** (o == **this**)  **returntrue**;  **if** (o**instanceof**Map.Entry) {  Map.Entry<?,?>e = (Map.Entry<?,?>)o;  **if** (Objects.*equals*(key, e.getKey()) &&  Objects.*equals*(value, e.getValue()))  **returntrue**;  }  **returnfalse**;  }  }  -----  } |

As you see, this inner class has four fields, key, value, next and hash.

**key :** It stores the key of an element and its final.

**value:** It holds the value of an element.

**next:**It holds the pointer to next key-value pair. This attribute makes the key-value pairs stored as a linked list. (the address of the next linked list)

**hash:**It holds the hashcode of the key.

These Entry objects are stored in an array called table[]. This array is initially of size 16. It is defined like below.

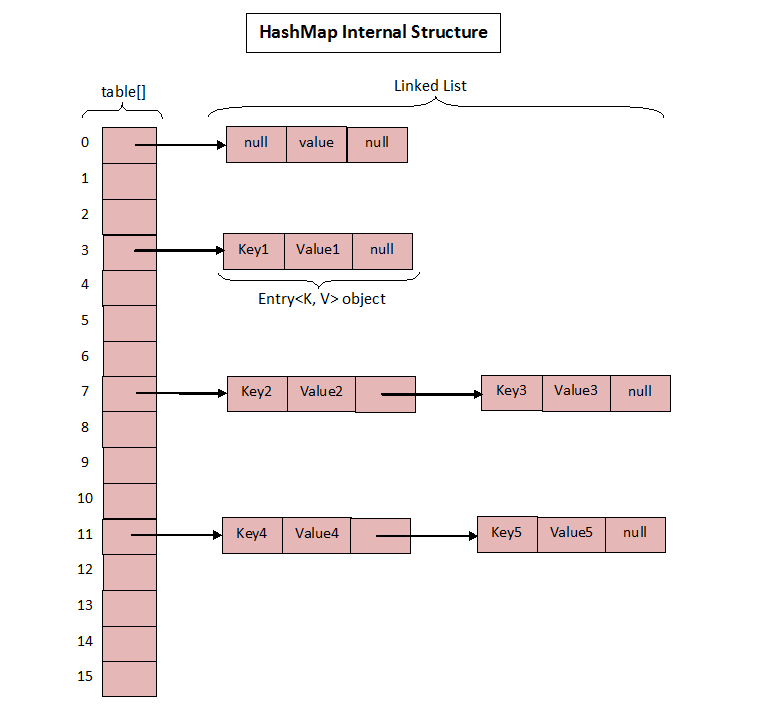
/\*\*

\* The table, resized as necessary. Length MUST Always be a power of two.

\*/

transient Entry<K,V>[] table;

To summarize the whole HashMap structure, object of Entry<K, Veach key-value pair is stored in an > class. This class has an attribute called next which holds the pointer to next key-value pair. This makes the key-value pairs stored as a linked list. All these Entry<K, V> objects are stored in an array called table[]. The below image best describes the HashMap structure.

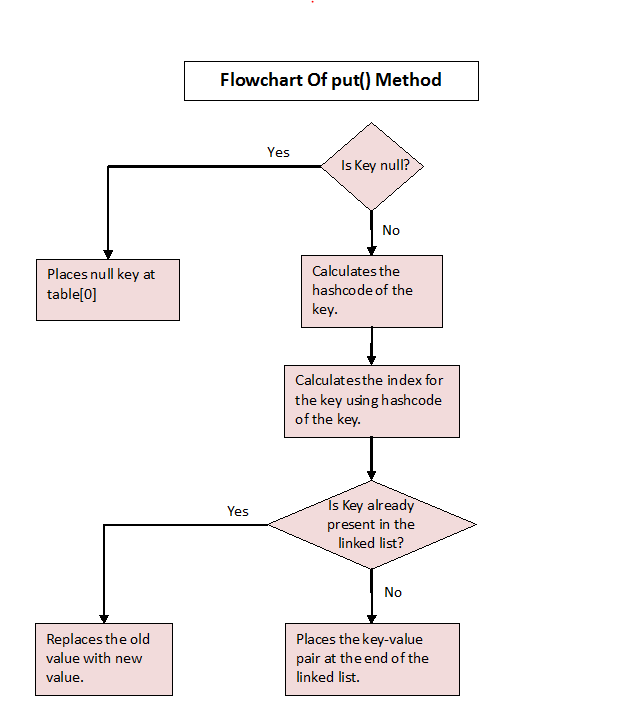


HashMap works on the principle of hashing.

Hash Map uses the hashcode() method to calculate a hash value. Hash value is calculated using the key object, the hash value is used to find the correct index [bucket] where entry object will be stored in table.

HashMap uses the equals() method to find the correct key whose value is to be retrieved in case of get() and to find if that key already exists or not in case of put().

Hashing Collision (crash) means more than one key having same hash value, in that case Entry objects are stored as a linked list with in a same bucket.



Step 1 : First checks whether the key is null or not. If the key is null, it calls *putForNullKey()* method. *table[0]* is always reserved for nssull key. Because, hash code of null is 0.

Step 2 : If the key is not null, then it calculates the hash code of the key by calling *hash()* method.

Step 3 : Calls *indexFor()* method by passing the hash code calculated in step 2 and length of the *table[]* array. This method returns index in *table[]* array for the specified key-value pair.

Step 4 : After getting the index, it checks all keys present in the linked list at that index ( or bucket). If the key is already present in the linked list, it replaces the old value with new value.

Step 5 : If the key is not present in the linked list, it appends the specified key-value pair at the end of the linked list.

**Get method works internally:**

Step 1 : First checks whether specified key is null or not. If the key is null, it calls *getForNullKey()* method.

Step 2 : If the key is not null, hash code of the specified key is calculated.

Step 3 : *indexFor()* method is used to find out the index of the specified key in the *table[]* array.

Step 4 : After getting index, it will iterate though linked list at that position and checks for the key using equals() method. If the key is found, it returns the value associated with it. otherwise returns null.

**Java 8 Improvements of Hashmap:**

The way [java.util.HashMap](https://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html) entries are indexed and stored has changed in the Java 8 update. Hash elements use balanced trees instead of linked lists under certain circumstances now. All these is what this article is about.

The main idea is that when the number of items in a hash is larger than a certain value, the hash will change from using a linked list of elements or entries to a balanced tree, this will improve the worst case performance from O(n) to O(log n)

**How linked list is replaced with binary tree?:**

In Java 8, HashMap replaces linked list with a binary tree when the number of elements in a bucket reaches certain threshold. While converting the list to binary tree, hashcode is used as a branching variable. If there are two different hashcodes in the same bucket, one is considered bigger and goes to the right of the tree and other one to the left. But when both the hashcodes are equal, HashMap assumes that the keys are comparable, and compares the key to determine the direction so that some order can be maintained. It is a good practice to make the keys of HashMap comparable.

Java 8 hash elements use balanced trees instead of linked lists after a certain threshold is reached. Which means HashMap starts with storing Entry objects in linked list, but after the number of items in a hash becomes larger than a certain threshold, the hash will change from using a linked list to a balanced tree. This improves the worst case performance from**O(n)** to **O(log n)**.

**Java 8 Date:**

1. Immutability: All the classes in the new Date Time API are immutable and good for multithreaded environments.
2. Separation of Concerns: The new API separates clearly between human readable date time and machine time (unix timestamp). It defines separate classes for Date, Time, DateTime, Timestamp, Timezone etc.
3. Clarity: The methods are clearly defined and perform the same action in all the classes. For example, to get the current instance we have now() method. There are format() and parse() methods defined in all these classes rather than having a separate class for them.

|  |
| --- |
| LocalDate:  //Current Date  LocalDatetoday = LocalDate.*now*();  System.***out***.println("Current Date="+today);    //Creating LocalDate by providing input arguments  LocalDatefirstDay\_2014 = LocalDate.*of*(2014, Month.***JANUARY***, 1);  System.***out***.println("Specific Date="+firstDay\_2014);      LocalDatetodayKolkata = LocalDate.*now*(ZoneId.*of*("Asia/Kolkata"));  System.***out***.println("Current Date in IST="+todayKolkata);  **LocalTime:**  //Current Time  LocalTimetime = LocalTime.*now*();  System.***out***.println("Current Time="+time);    //Creating LocalTime by providing input arguments  LocalTimespecificTime = LocalTime.*of*(12,20,25,40);  System.***out***.println("Specific Time of Day="+specificTime);      //Try creating time by providing invalid inputs  //LocalTimeinvalidTime = LocalTime.of(25,20);  //Exception in thread "main" java.time.DateTimeException:  //Invalid value for HourOfDay (valid values 0 - 23): 25    //Current date in "Asia/Kolkata", you can get it from ZoneIdjavadoc  LocalTimetimeKolkata = LocalTime.*now*(ZoneId.*of*("Asia/Kolkata"));  System.***out***.println("Current Time in IST="+timeKolkata);  //LocalDateTime  LocalDateTimetoday = LocalDateTime.*now*();  System.***out***.println("Current DateTime="+today);    //Current Date using LocalDate and LocalTime  today = LocalDateTime.*of*(LocalDate.*now*(), LocalTime.*now*());  System.***out***.println("Current DateTime="+today);    //Creating LocalDateTime by providing input arguments  LocalDateTimespecificDate = LocalDateTime.*of*(2014, Month.***JANUARY***, 1, 10, 10, 30);  System.***out***.println("Specific Date="+specificDate);  LocalDated = LocalDate.*now*();  System.***out***.println(d);  Stringstr =d.format(DateTimeFormatter.*ofPattern*("d:MMM::uuuu"));  System.***out***.println(str); |

**Generic Type:**

Generics was added in Java 5 to provide compile-time type checking and removing risk of ClassCastException that was common while working with collection classes.

Advantage of Java Generics

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

1. Type-safety: We can hold only a single type of objects in generics. It doesn’t allow to store other objects.
2. Type casting is not required: There is no need to typecast the object.
3. Compile-Time Checking: It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than runtime.
4. Code Reuse: We can write a method/class/interface once and use for any type we want.

|  |
| --- |
| Before Generics, we need to type cast.  List list = new ArrayList();  list.add("hello");  String s = (String) list.get(0);//typecasting  After Generics, we don't need to typecast the object.  List<String> list = new ArrayList<String>();  list.add("hello");  String s = list.get(0);  List<String> list = new ArrayList<String>();  list.add("hello");  list.add(32);//Compile Time Error  Syntax to use generic collection  ClassOrInterface<Type>  Example to use Generics in java  ArrayList<String>  **interface**DemoInterFace<T> {  **public** T add(T t1, T t2);  }  **class**StringExample**implements**DemoInterFace<String> {  @Override  **public** String add(String t1, String t2) {  **return**t1 + t2;  }  }  **class**IntegerExample**implements**DemoInterFace<Integer> {  @Override  **public** Integer add(Integer t1, Integer t2) {  // **TODO** Auto-generated method stub  **return**t1 + t2;  }  }  **class**GenericClass<T> {  **private** T t;  **private** T t1;  **private**T[] arr;  **public** T getT1() {  **return**t1;  }  **publicvoid** setT1(T t1) {  **this**.t1 = t1;  }  **public**T[] getArr() {  **return**arr;  }  **publicvoid**setArr(T[] arr) {  **this**.arr = arr;  }  **public**GenericClass<T>getG() {  **return**g;  }  **publicvoid**setG(GenericClass<T>g) {  **this**.g = g;  }  **private**GenericClass<T>g;  **public** T getT() {  **return**t;  }  **publicvoid**setT(T t) {  **this**.t = t;  }  **publicvoid**displayArry(T[] elements) {  Arrays.*asList*(elements).stream().forEach(System.***out***::println);  }  }  **publicclass**GenericType {  **publicstaticvoid**main(String[] args) {  GenericClassg = **new**GenericClass();  g.setT(10);  System.***out***.println(g.getT());  String[] str = { "scala", "java", "spark" };  g.displayArry(str);  Integer[] intr = { 1, 10, 20 };  g.displayArry(intr);  g.setArr(str);  g.getArr();  Arrays.*asList*(g.getArr()).stream().forEach(System.***out***::println);  GenericClass<String>g1 = **new**GenericClass<String>();  g1.setT("nag");  System.***out***.println(g1.getT());  DemoInterFacedemoInterFace1 = **new**StringExample();  System.***out***.println(demoInterFace1.add("nagendra", "mekala"));  DemoInterFacedemoInterFace2 = **new**IntegerExample();  System.***out***.println(demoInterFace2.add(10, 2));  }  } |

**Stack:**

Stack can be simply defined as Last In First Out (LIFO) data structure,i.e.,the last element added at the top of the stack(In) should be the first element to be removed(Out) from the stack.

Whatever are element added in last we can access first(LIFO).

**Queue:**

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.

|  |
| --- |
| **publicclass**MyStackImpl {  **privateint**stackSize;  **privateint**[] stackArr;  **privateint**top;    /\*\*  \* constructor to create stack with size  \* **@param** size  \*/  **public**MyStackImpl(**int**size) {  **this**.stackSize = size;  **this**.stackArr = **newint**[stackSize];  **this**.top = -1;  }    /\*\*  \* This method adds new entry to the top  \* of the stack  \* **@param**entry  \* **@throws** Exception  \*/  **publicvoid**push(**int**entry) **throws** Exception {  **if**(**this**.isStackFull()){  **thrownew**Exception("Stack is already full. Can not add element.");  }  System.***out***.println("Adding: "+entry);  **this**.stackArr[++top] = entry;  }    /\*\*  \* This method removes an entry from the  \* top of the stack.  \* **@return**  \* **@throws** Exception  \*/  **publicint**pop() **throws** Exception {  **if**(**this**.isStackEmpty()){  **thrownew**Exception("Stack is empty. Can not remove element.");  }  **int**entry = **this**.stackArr[top--];    System.***out***.println("Removed entry: "+entry);  **return**entry;  }    /\*\*  \* This method returns top of the stack  \* without removing it.  \* **@return**  \*/  **publicint**peek() {  **return**stackArr[top];  }    /\*\*  \* This method returns true if the stack is  \* empty  \* **@return**  \*/  **publicboolean**isStackEmpty() {  **return** (top == -1);  }    /\*\*  \* This method returns true if the stack is full  \* **@return**  \*/  **publicboolean**isStackFull() {  **return** (top == stackSize - 1);  }    **publicstaticvoid**main(String[] args) {  MyStackImplstack = **new**MyStackImpl(5);  **try** {  stack.push(4);  stack.push(8);  stack.push(3);  stack.push(89);  stack.pop();  stack.push(34);  stack.push(45);  stack.push(78);  } **catch** (Exception e) {  System.***out***.println(e.getMessage());  }  **try** {  stack.pop();  stack.pop();  stack.pop();  stack.pop();  stack.pop();  stack.pop();  } **catch** (Exception e) {  System.***out***.println(e.getMessage());  }  }  } |

**Files I/O**

## Stream

A stream is a sequence of data.In Java a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

In java, 3 streams are created for us automatically. All these streams are attached with console.

**1) System.out:**standard output stream

**2) System.in:**standard input stream

**3) System.err:**standard error stream

## OutputStream

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

## InputStream

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

## OutputStream class

OutputStream class is an abstract class. It is the super class of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

## Useful methods of OutputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public void write(int)throws IOException | is used to write a byte to the current output stream. |
| 2) public void write(byte[])throws IOException | is used to write an array of byte to the current output stream. |
| 3) public void flush()throws IOException | flushes the current output stream. |
| 4) public void close()throws IOException | is used to close the current output stream. |

## OutputStream Hierarchy

Java output stream hierarchy

## InputStream class

InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

## Useful methods of InputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public abstract int read()throws IOException | reads the next byte of data from the input stream. It returns -1 at the end of file. |
| 2) public int available()throws IOException | returns an estimate of the number of bytes that can be read from the current input stream. |
| 3) public void close()throws IOException | is used to close the current input stream. |

## InputStream Hierarchy

Java input stream hierarchy

# **Java BufferedOutputStream Class**

Java BufferedOutputStream [class](https://www.javatpoint.com/object-and-class-in-java) is used for buffering an output stream. It internally uses buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

For adding the buffer in an OutputStream, use the BufferedOutputStream class. Let's see the syntax for adding the buffer in an OutputStream.

**OutputStreamos= new BufferedOutputStream(new FileOutputStream("D:\\IO Package\\testout.txt"));**

Java BufferedOutputStream class constructors

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedOutputStream(OutputStreamos) | It creates the new buffered output stream which is used for writing the data to the specified output stream. |
| BufferedOutputStream(OutputStreamos, int size) | It creates the new buffered output stream which is used for writing the data to the specified output stream with a specified buffer size. |

## Java BufferedOutputStream class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| void write(int b) | It writes the specified byte to the buffered output stream. |
| void write(byte[] b, int off, int len) | It write the bytes from the specified byte-input stream into a specified byte [array](https://www.javatpoint.com/array-in-java), starting with the given offset |
| void flush() | It flushes the buffered output stream. |

## Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the [FileOutputStream](https://www.javatpoint.com/java-fileoutputstream-class)[object](https://www.javatpoint.com/object-and-class-in-java). The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

|  |
| --- |
| 1. **package** com.javatpoint; 2. **import** java.io.\*; 3. **public** **class** BufferedOutputStreamExample{ 4. **public** **static** **void** main(String args[])**throws** Exception{ 5. FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt"); 6. BufferedOutputStream bout=**new** BufferedOutputStream(fout); 7. String s="Welcome to javaTpoint."; 8. **byte** b[]=s.getBytes(); 9. bout.write(b); 10. bout.flush(); 11. bout.close(); 12. fout.close(); 13. System.out.println("success"); 14. } 15. } |

# **Java BufferedInputStream Class**

Java BufferedInputStream [class](https://www.javatpoint.com/object-and-class-in-java) is used to read information from [stream](https://www.javatpoint.com/java-8-stream). It internally uses buffer mechanism to make the performance fast.

The important points about BufferedInputStream are:

* When the bytes from the stream are skipped or read, the internal buffer automatically refilled from the contained input stream, many bytes at a time.
* When a BufferedInputStream is created, an internal buffer [array](https://www.javatpoint.com/array-in-java) is created.

|  |
| --- |
| 1. **import** java.io.\*; 2. **public** **class** BufferedInputStreamExample{ 3. **public** **static** **void** main(String args[]){ 4. **try**{ 5. FileInputStream fin=**new** FileInputStream("D:\\testout.txt"); 6. BufferedInputStream bin=**new** BufferedInputStream(fin); 7. **int** i; 8. **while**((i=bin.read())!=-1){ 9. System.out.print((**char**)i); 10. } 11. bin.close(); 12. fin.close(); 13. }**catch**(Exception e){System.out.println(e);} 14. } 15. } |

**FileWriter:**FileWriter is the simplest way to write a file in java, it provides overloaded write method to write int, byte array and String to the File. You can also write part of the String or byte array using FileWriter. FileWriter writes directly into Files and should be used only when number of writes are less.

**BufferedWriter:**BufferedWriter is almost similar toFileWriter but it uses internal buffer to write data into File. So if the number of write operations are more, the actual IO operations are less and performance is better. You should use BufferedWriter when number of write operations are more.

**FileOutputStream:**FileWriter and BufferedWriter are meant to write text to the file but when you need raw stream data to be written into file, you should use FileOutputStream to write file in java.

**Files:** Java 7 introduced Files utility class and we can write a file using it’s write function, internally it’s using OutputStream to write byte array into file.

# **Java BufferedWriter Class**

Java BufferedWriter class is used to provide buffering for Writer instances. It makes the performance fast. It inherits [Writer](https://www.javatpoint.com/java-writer-class) class. The buffering characters are used for providing the efficient writing of single [arrays](https://www.javatpoint.com/array-in-java), characters, and [strings](https://www.javatpoint.com/java-string).

**Class constructors**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedWriter(Writer wrt) | It is used to create a buffered character output stream that uses the default size for an output buffer. |
| BufferedWriter(Writer wrt, int size) | It is used to create a buffered character output stream that uses the specified size for an output buffer. |

**Class methods**

|  |  |
| --- | --- |
| **Method** | **Description** |
| void newLine() | It is used to add a new line by writing a line separator. |
| void write(int c) | It is used to write a single character. |
| void write(char[] cbuf, int off, int len) | It is used to write a portion of an array of characters. |
| void write(String s, int off, int len) | It is used to write a portion of a string. |
| void flush() | It is used to flushes the input stream. |
| void close() | It is used to closes the input stream |

|  |
| --- |
| 1. **package** com.javatpoint; 2. **import** java.io.\*; 3. **public** **class** BufferedWriterExample { 4. **public** **static** **void** main(String[] args) **throws** Exception { 5. FileWriter writer = **new** FileWriter("D:\\testout.txt"); 6. BufferedWriter buffer = **new** BufferedWriter(writer); 7. buffer.write("Welcome to javaTpoint."); 8. buffer.close(); 9. System.out.println("Success");    }  } |

# **Java BufferedReader Class**

Java BufferedReader class is used to read the text from a character-based input stream. It can be used to read data line by line by readLine() method. It makes the performance fast. It inherits [Reader](https://www.javatpoint.com/java-reader-class) [class](https://www.javatpoint.com/object-and-class-in-java).

**Java BufferedReader class constructors**

|  |  |
| --- | --- |
| **Constructor** | **Description** |
| BufferedReader(Reader rd) | It is used to create a buffered character input stream that uses the default size for an input buffer. |
| BufferedReader(Reader rd, int size) | It is used to create a buffered character input stream that uses the specified size for an input buffer. |

## Java BufferedReader class methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| int read() | It is used for reading a single character. |
| int read(char[] cbuf, int off, int len) | It is used for reading characters into a portion of an [array](https://www.javatpoint.com/array-in-java). |
| booleanmarkSupported() | It is used to test the input stream support for the mark and reset method. |
| String readLine() | It is used for reading a line of text. |
| booleanready() | It is used to test whether the input stream is ready to be read. |
| long skip(long n) | It is used for skipping the characters. |
| void reset() | It repositions the [stream](https://www.javatpoint.com/java-8-stream) at a position the mark method was last called on this input stream. |
| void mark(int readAheadLimit) | It is used for marking the present position in a stream. |
| void close() | It closes the input stream and releases any of the system resources associated with the stream. |

|  |
| --- |
| 1. Ex: 1 2. FileReader fr=**new** FileReader("D:\\testout.txt"); 3. BufferedReader br=**new** BufferedReader(fr); 4. **int** i; 5. **while**((i=br.read())!=-1){ 6. System.out.print((**char**)i); 7. } 8. br.close(); 9. fr.close(); 10. Ex: 2 11. InputStreamReader r=**new** InputStreamReader(System.in); 12. BufferedReader br=**new** BufferedReader(r); 13. System.out.println("Enter your name"); 14. String name=br.readLine(); 15. System.out.println("Welcome "+name); |

# **java.nio.file.Files Class**

Java Files class was introduced in Java 1.7 and is a part of java.nio.file package.

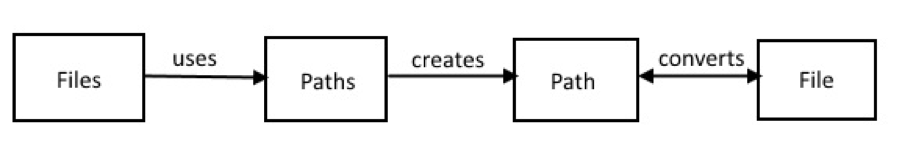
* Java Files class contains [static](https://www.journaldev.com/1365/static-keyword-in-java) methods that work on files and directories.
* This class is used for basic file operations like create, read, write, copy and delete the files or directories of the file system.

Before move ahead let’s have a look at the below terms first:

1. **Path**: This is the interface that replaces java.io.File class as the representation of a file or a directory when we work in Java NIO.
2. **Paths**: This class contains a static method to create Path instance.

Creating a new [Path](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Path.html)object doesn’t create a file on the file system. A Path object just indicates a file/directory or any other resource location. The resource might or might not be present in the identified location.

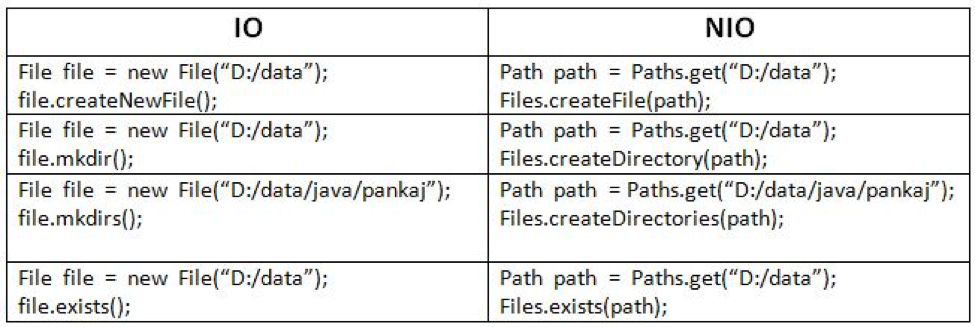
java.nio.file.Path [interface](https://www.journaldev.com/1601/interface-in-java) is just like the old [java.io.File](https://www.journaldev.com/17330/java-file-class-java-io-file) class. Path represents location of the file and when we create a Path to new file, it does not create actual file until we create it using Files.createFile(Path filePath).



As we can see in above diagram, Paths class is used to create instance of Path and Files class uses Path instance to work on a file.

**File** and Path objects know how to convert to the other, that’s how we can use older code to interact with new **Files** utility.

### Java IO vs NIO



The advantage of using the Java Files API is that the code is more readable than before and also the parts of the code encapsulated by the API can be reusable.

|  |
| --- |
| **import**java.io.BufferedReader;  **import**java.io.BufferedWriter;  **import**java.io.IOException;  **import**java.nio.charset.Charset;  **import**java.nio.file.Files;  **import**java.nio.file.Path;  **import**java.nio.file.Paths;  **publicclass**NewFilesApiTest {  **publicstaticvoid**main(String[] args) {  //Creating a new file  Path newFile = Paths.*get*("test1");  **try** {  Files.*deleteIfExists*(newFile);  newFile = Files.*createFile*(newFile);  } **catch** (IOExceptionex) {  System.***out***.println("Error creating file");  }  System.***out***.println(Files.*exists*(newFile));  //Writing to file  **try**(BufferedWriterwriter = Files.*newBufferedWriter*(  newFile, Charset.*defaultCharset*())){  writer.append("This is first line");  writer.newLine();  writer.append("This is second line");  writer.newLine();  writer.append("This is third line");  writer.newLine();  writer.flush();  }**catch**(IOExceptionexception){  System.***out***.println("Error writing to file");  }  //Reading from file  **try**(BufferedReaderreader = Files.*newBufferedReader*(  newFile, Charset.*defaultCharset*())){  String lineFromFile = "";  System.***out***.println("The contents of file are: ");  **while**((lineFromFile = reader.readLine()) != **null**){  System.***out***.println(lineFromFile);  }  }**catch**(IOExceptionexception){  System.***out***.println("Error while reading file");  }  }} |

# **Serialization in Java**

Serialization in Java allows us to convert an Object to stream that we can send over the network or save it as file or store in DB for later usage. Deserialization is the process of converting Object stream to actual Java Object to be used in our program

**Serialization in java** is a mechanism of writing the state of an object into a byte stream.

It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

The reverse operation of serialization is called deserialization.

**Object Serialization** is a process used to convert the state of an object into a byte stream, which can be persisted into disk/file or sent over the network to any other running Java virtual machine. The reverse process of creating an object from the byte stream is called **deserialization**. The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

What is use of marker interface

Marker Interfaces are used to indicate something to compiler/JVM. If JVM see that a class is a object of Marker Interface then it will perform some special operation. Take an example with Serializable, Clonnable marker interface, if JVM see a Class is Serialized/Clonnable then It will do some special operation on it, similar way if JVM sees one Class is implemented custom marker interface which is created by our self then the JVM do some special operation. How it’d do the special operati

### **Advantage of Java Serialization**

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

## java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" java classes so that objects of these classes may get certain capability. The Cloneable and Remote are also marker interfaces.

It must be implemented by the class whose object you want to persist.

The String class and all the wrapper classes implements java.io.Serializable interface by default.

## Deserialization in java

Deserialization is the process of reconstructing the object from the serialized state.It is the reverse operation of serialization.

## ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

## Examples

In this example, we are going to serialize the object of Student class. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

|  |
| --- |
| 1. **import** java.io.\*; 2. **class** Persist{ 3. **public** **static** **void** main(String args[])**throws** Exception{ 4. Student s1 =**new** Student(211,"ravi"); 6. FileOutputStream fout=**new** FileOutputStream("f.txt"); 7. ObjectOutputStream out=**new** ObjectOutputStream(fout); 9. out.writeObject(s1); 10. out.flush(); 11. System.out.println("success"); 12. } 13. } 14. **import** java.io.\*; 15. **class** Depersist{ 16. **public** **static** **void** main(String args[])**throws** Exception{ 18. ObjectInputStream in=**new** ObjectInputStream(**new** FileInputStream("f.txt")); 19. Student s=(Student)in.readObject(); 20. System.out.println(s.id+" "+s.name); 22. in.close(); 23. } 24. } |

## Java Serialization with Inheritance (IS-A Relationship)

If a class implements serializable then all its sub classes will also be serializable

## Java Serialization with Aggregation (HAS-A Relationship)

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

|  |
| --- |
| 1. Example 1: 2. **import** java.io.Serializable; 3. **class** Person **implements** Serializable{ 4. **int** id; 5. String name; 6. Person(**int** id, String name) { 7. **this**.id = id; 8. **this**.name = name; 9. } 10. } 11. **class** Student **extends** Person{ 12. String course; 13. **int** fee; 14. **public** Student(**int** id, String name, String course, **int** fee) { 15. **super**(id,name); 16. **this**.course=course; 17. **this**.fee=fee; 18. } 19. }   Now you can serialize the Student class object that extends the Person class which is Serializable.Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.  Example 2:   1. **class** Address{ 2. String addressLine,city,state; 3. **public** Address(String addressLine, String city, String state) { 4. **this**.addressLine=addressLine; 5. **this**.city=city; 6. **this**.state=state; 7. } 8. }   import java.io.Serializable;  public class Student implements Serializable{  int id;  String name;  Address address;//HAS-A  public Student(int id, String name) {  this.id = id;  this.name = name;  }  }   Address is not Serializable, you can not serialize the instance of Student class. |

**Note: All the objects within an object must be Serializable.**

## Java Serialization with 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.

|  |
| --- |
| 1. **class** Employee **implements** Serializable{ 2. **int** id; 3. String name; 4. **static** String company="SSS IT Pvt Ltd";//it won't be serialized 5. **public** Student(**int** id, String name) { 6. **this**.id = id; 7. **this**.name = name; 8. } 9. } |

## 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.

The Externalizable interface provides two methods:

* **public void writeExternal(ObjectOutput out) throws IOException**
* **public void readExternal(ObjectInput in) throws IOException**

## Java Transient Keyword

If you don't want to serialize any data member of a class, you can mark it as transient.

**Java transient** keyword is used in serialization. If you define any data member as transient, it will not be serialized.

Let's take an example, I have declared a class as Student, it has three data members id, name and age. If you serialize the object, all the values will be serialized but I don't want to serialize one value, e.g. age then we can declare the age data member as transient.

### **Example of Java Transient Keyword**

In this example, we have created the two classes Student and PersistExample. The age data member of the Student class is declared as transient, its value will not be serialized.

If you deserialize the object, you will get the default value for transient variable.

|  |
| --- |
| 1. **import** java.io.Serializable; 2. **public** **class** Student **implements** Serializable{ 3. **int** id; 4. String name; 5. **transient** **int** age;//Now it will not be serialized 6. **public** Student(**int** id, String name,**int** age) { 7. **this**.id = id; 8. **this**.name = name; 9. **this**.age=age; 10. } 11. } |

**Deference b/w Serializable and Externalizable**

* **Serialization Responsibility**

The key difference here is how we handle the serialization process. When a class implements the java.io.Serializable interface, the JVM takes full responsibility for serializing the class instance. In case of Externalizable, it’s the programmer who should take care of the whole serialization and also deserialization process.

* **Use Case**

If we need to serialize the entire object, the Serializable interface is a better fit. On the other hand, for custom serialization, we can control the process using Externalizable.

* **Reading Order**

While using Externalizable, it’s mandatory to read all the field states in the exact order as they were written. Otherwise, we’ll get an exception.

For example, if we change the reading order of the code and name properties in the Country class, a java.io.EOFException will be thrown.

* **Reduce no of fields**

This difference between Serializable and Externalizable is performance. You cannot do much to improve performance of default serialization process except reducing number of fields to be serialized by using [transient](http://java67.blogspot.sg/2012/08/what-is-transient-variable-in-java.html)and [static keyword](http://javarevisited.blogspot.sg/2011/11/static-keyword-method-variable-java.html) but with Externalizable interface you have full control over Serialization process  
  
Read more: <http://www.java67.com/2012/10/difference-between-serializable-vs-externalizable-interface.html#ixzz5mUGpMGTh>

<http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/>

<https://www.boraji.com/java-8-longfunction-interface-example>