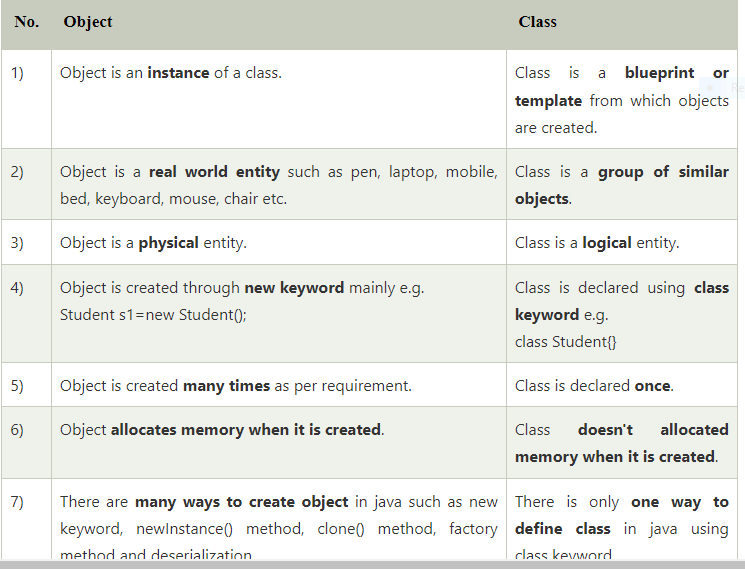
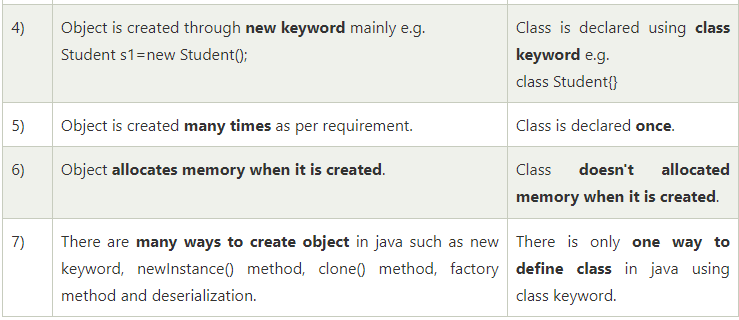
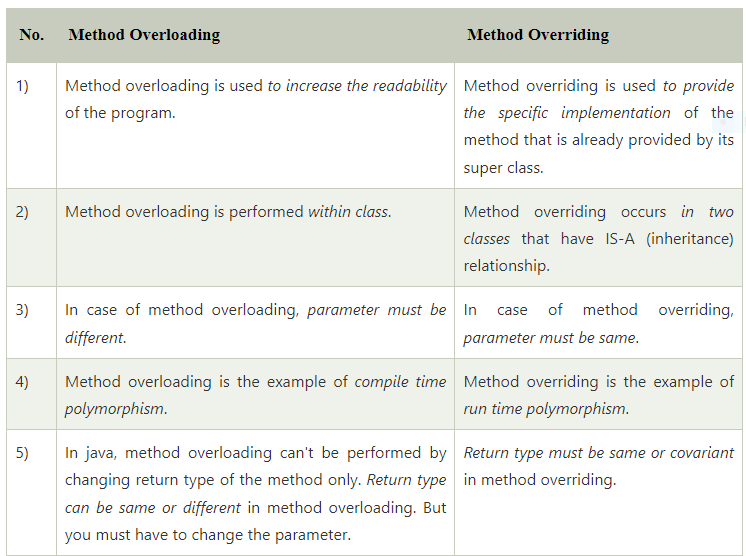
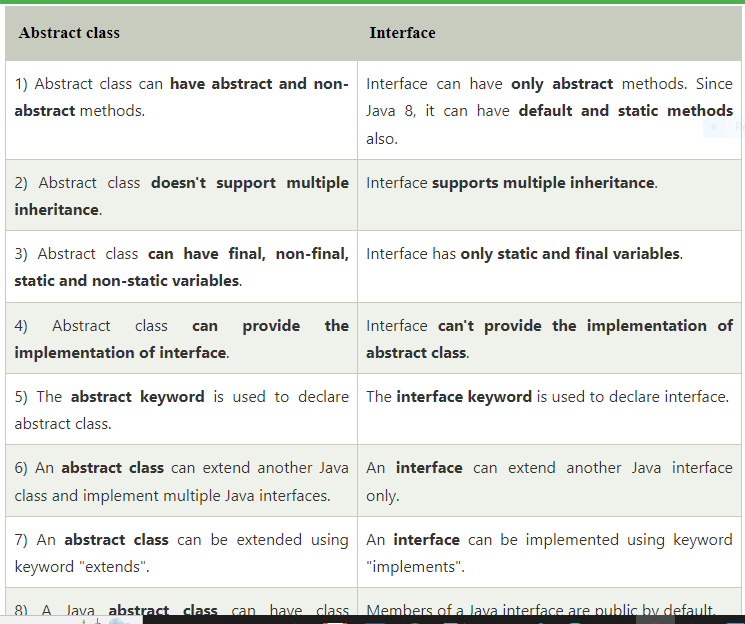
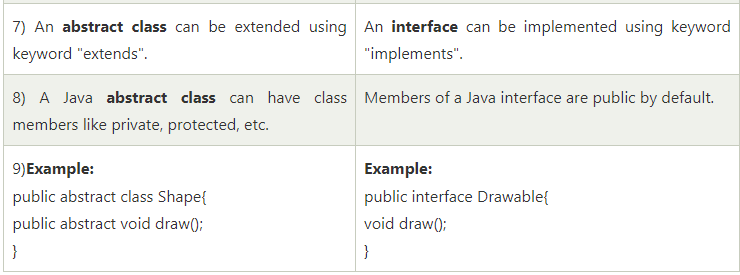
class vs object with example

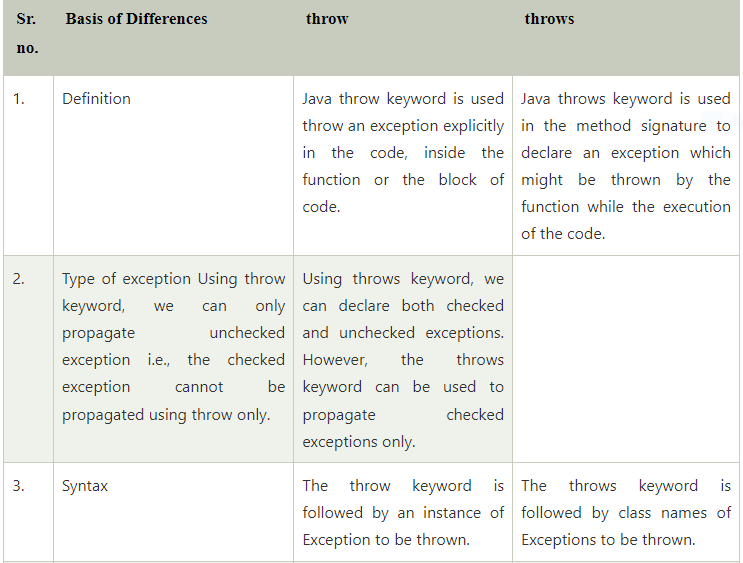


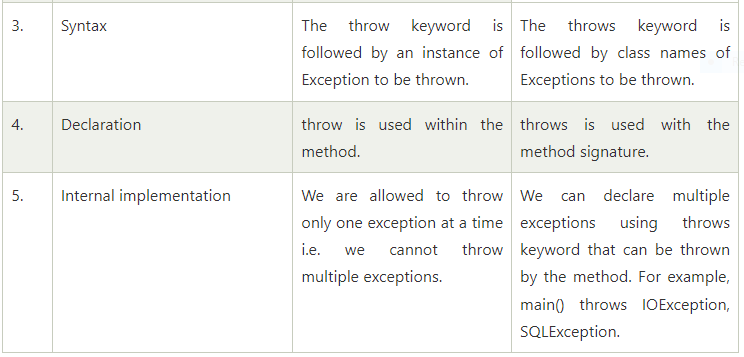


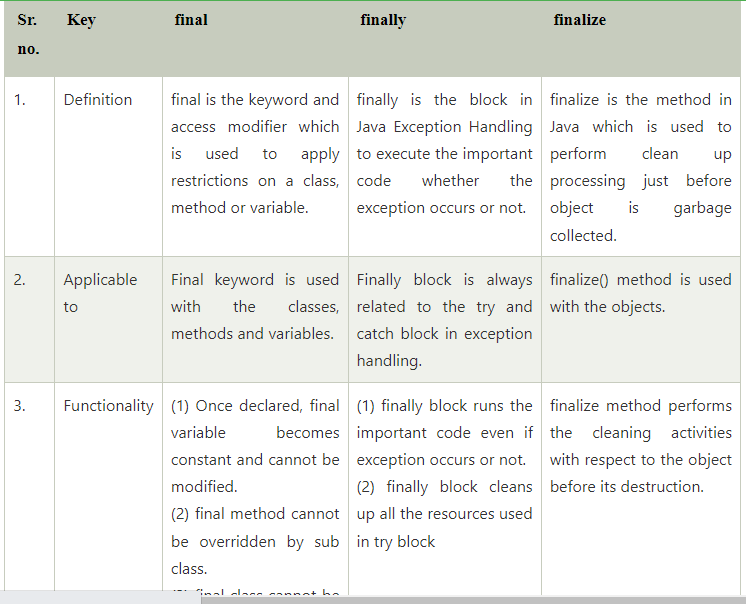


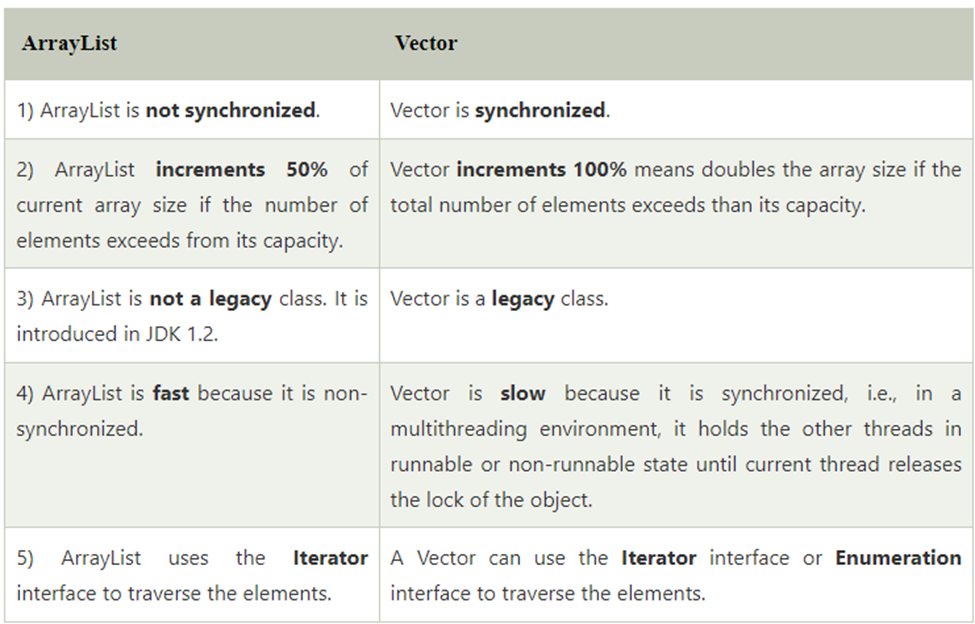
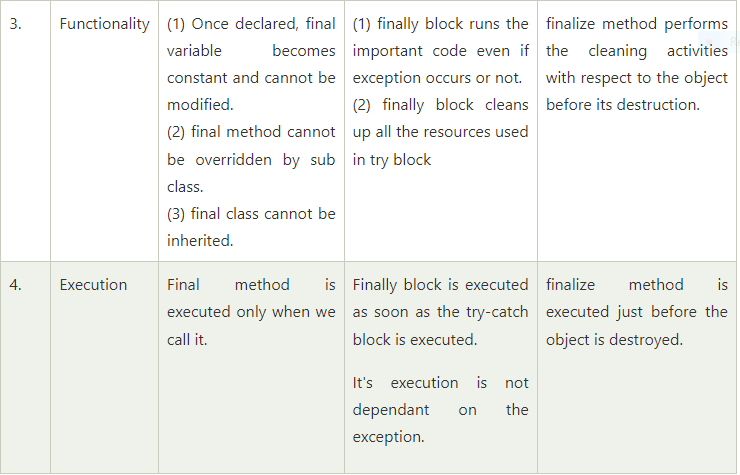


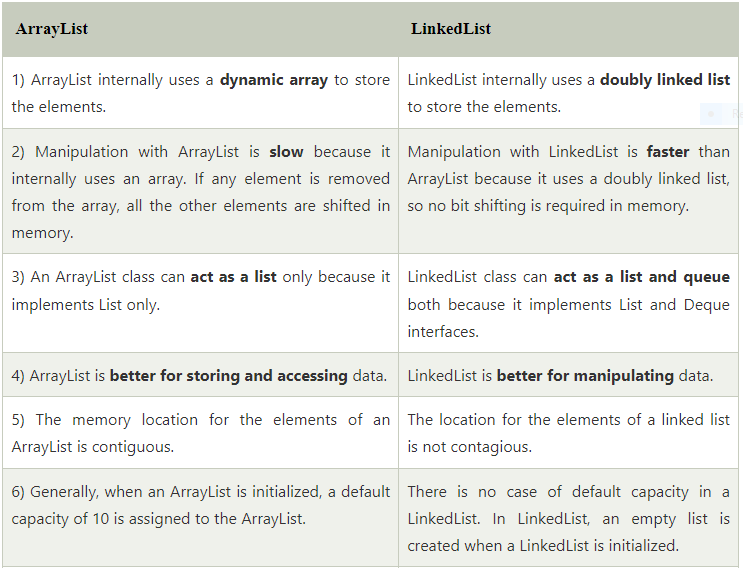


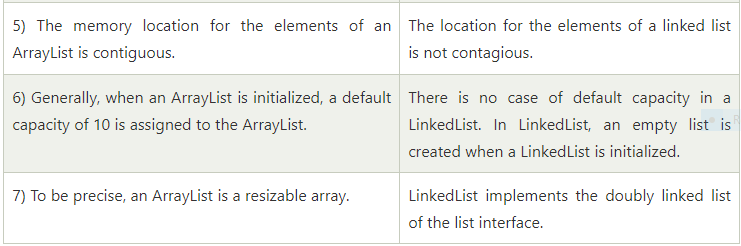


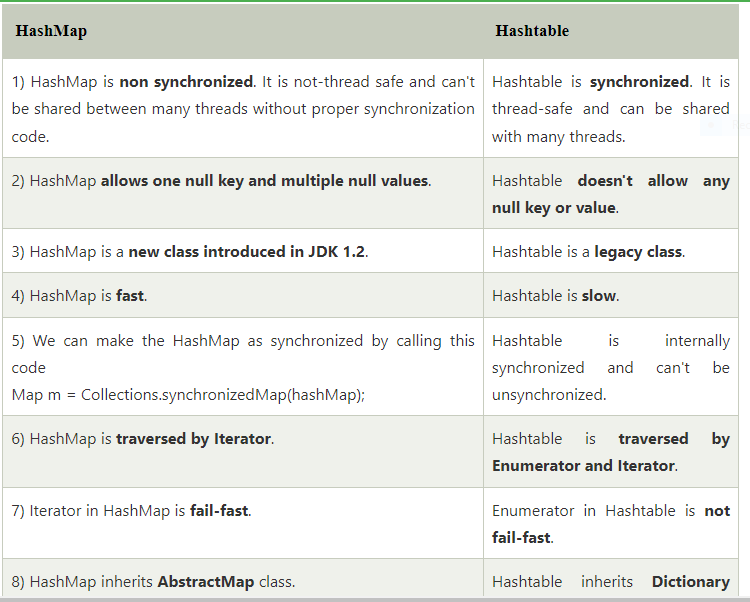




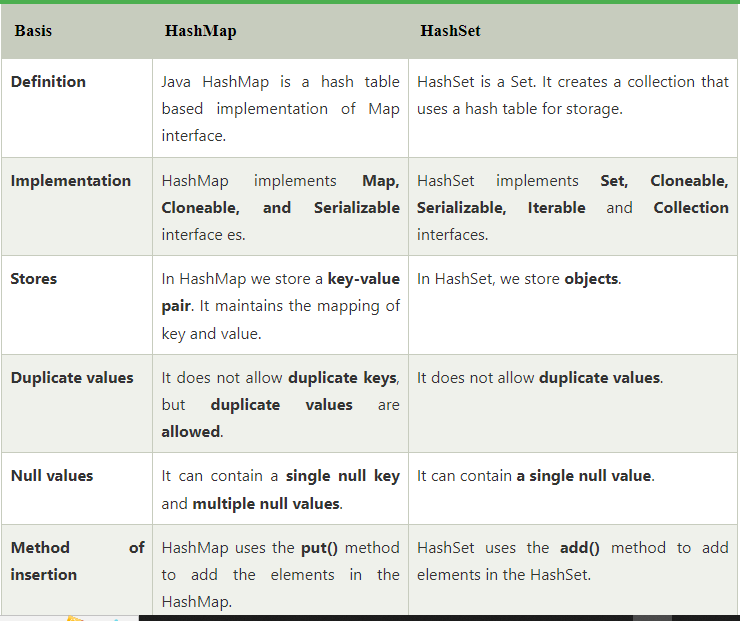


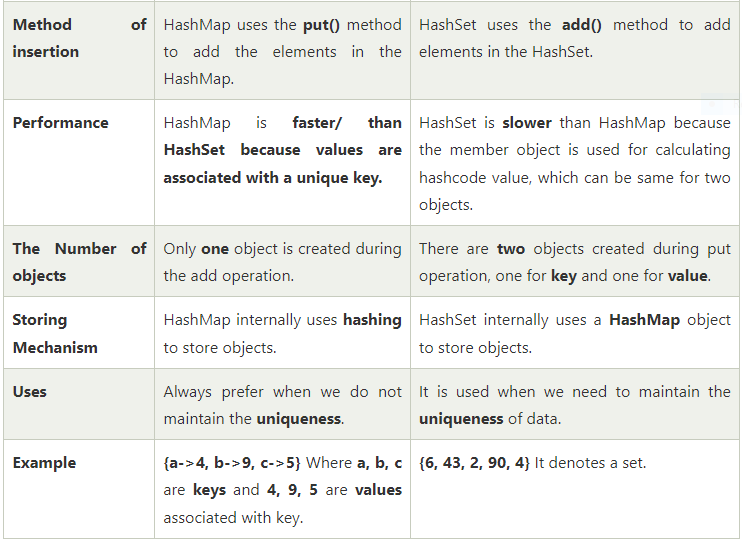


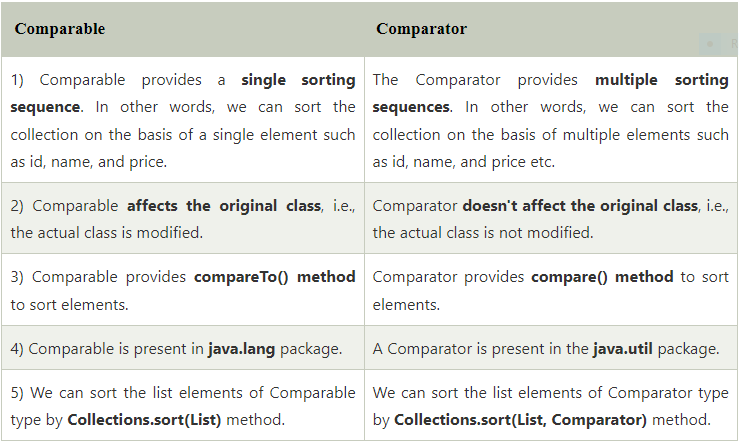


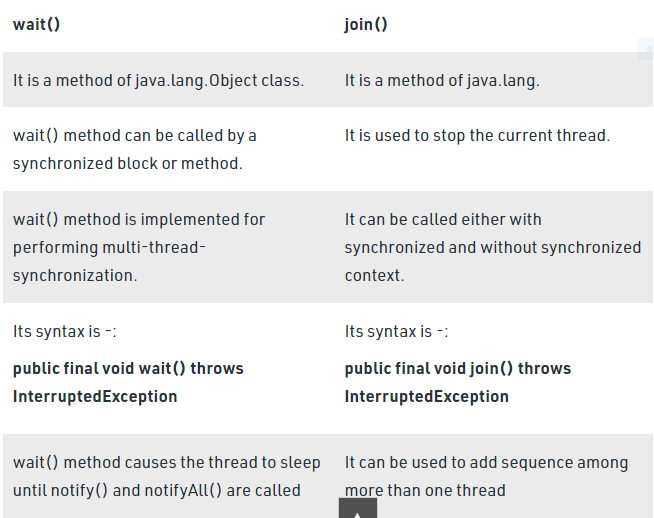


# Fail Fast and Fail Safe Iterators in Java

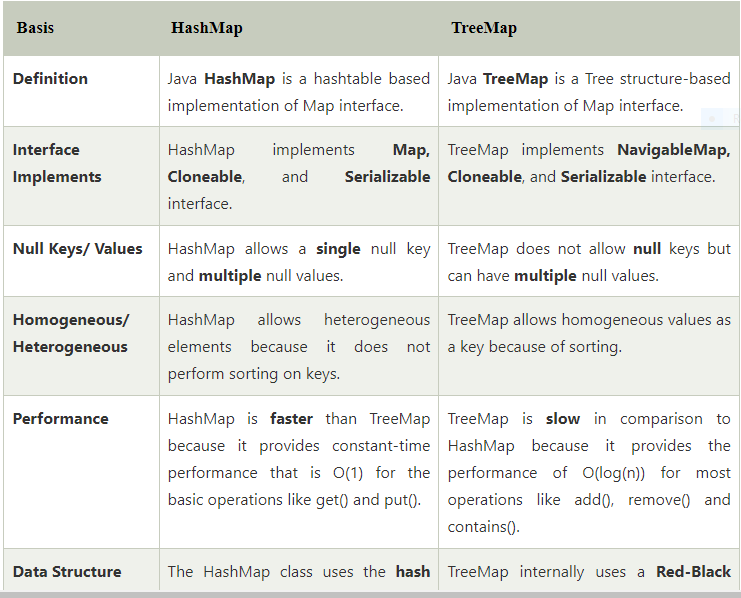


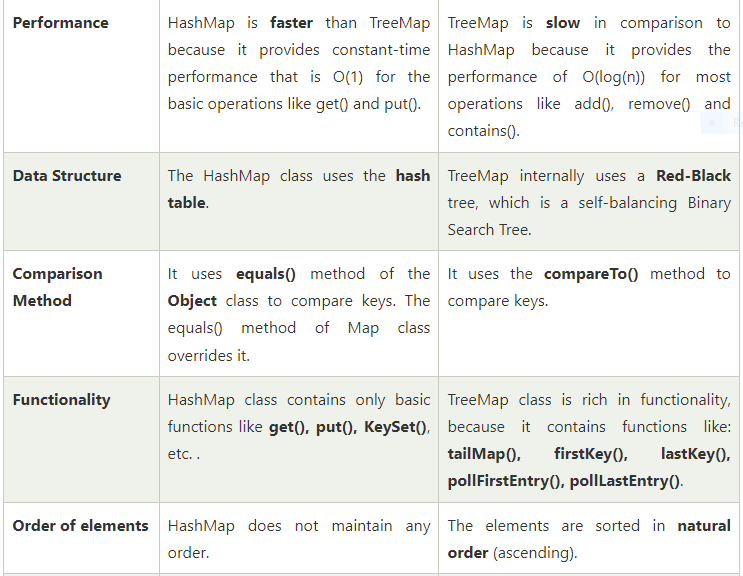


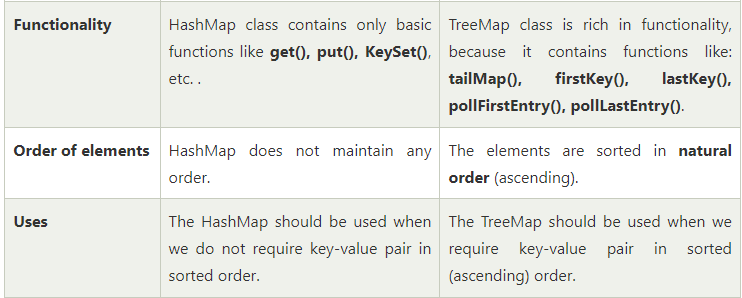


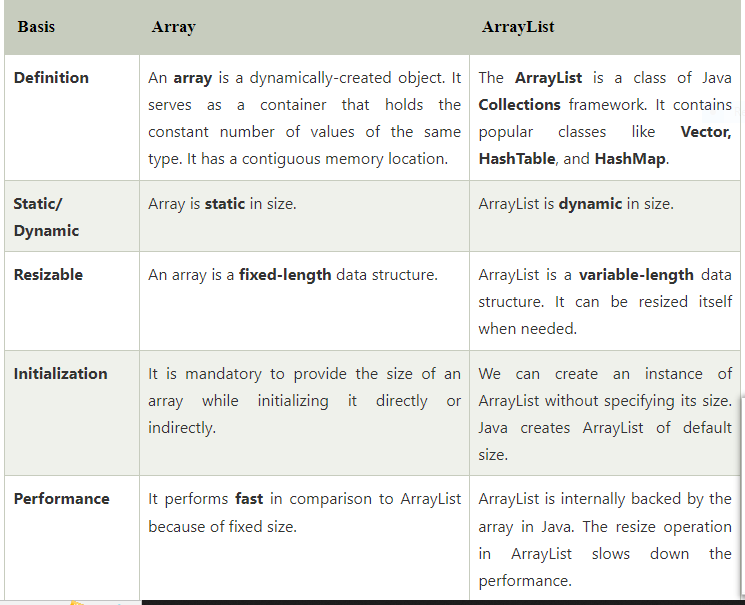


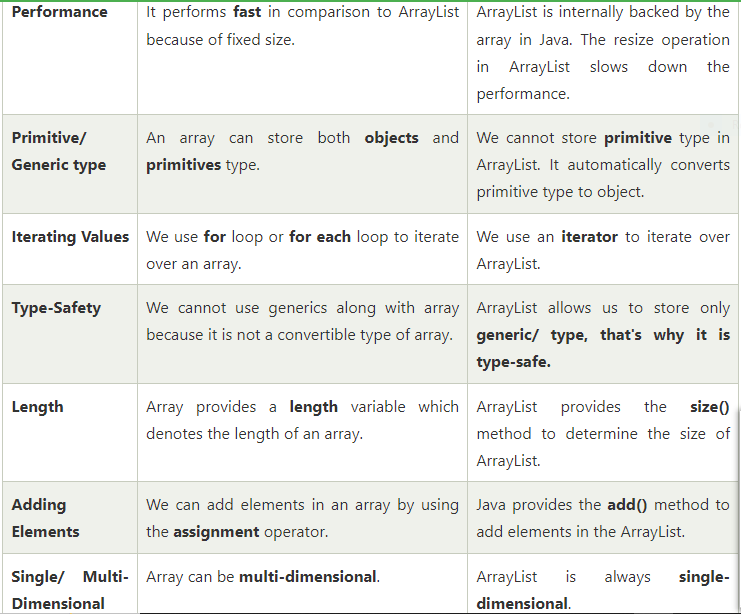
| **Collection** | **Collections** |
| --- | --- |
| It is an interface. | It is a utility class. |
| It is used to represent a group of individual objects as a single unit. | It defines several utility methods that are used to operate on collection. |
| The Collection is an interface that contains a static method since java8. The Interface can also contain abstract and default methods. | It contains only static methods. |

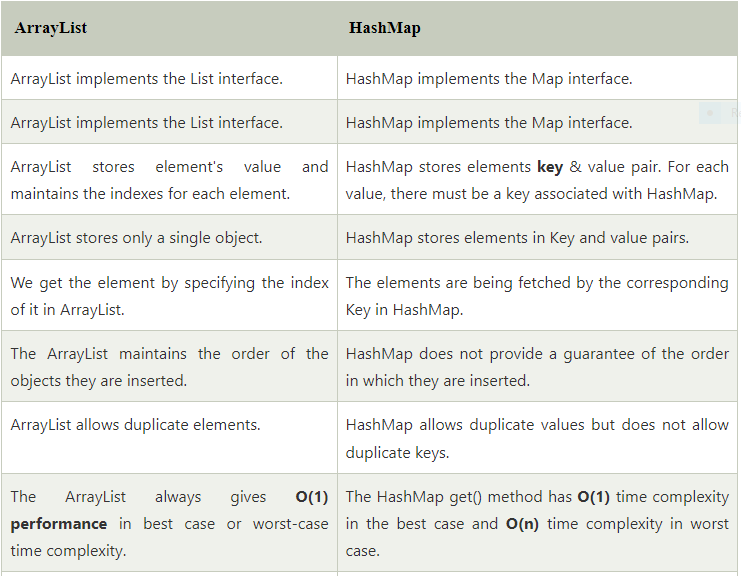


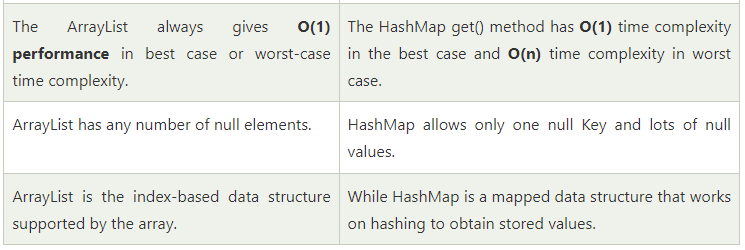












abstraction- real life example

**encapsulation -real life example**

**Polymorphism - real life example**

**lazy and eager resolution**

**what is class Class? When do you use it in Java Program?**

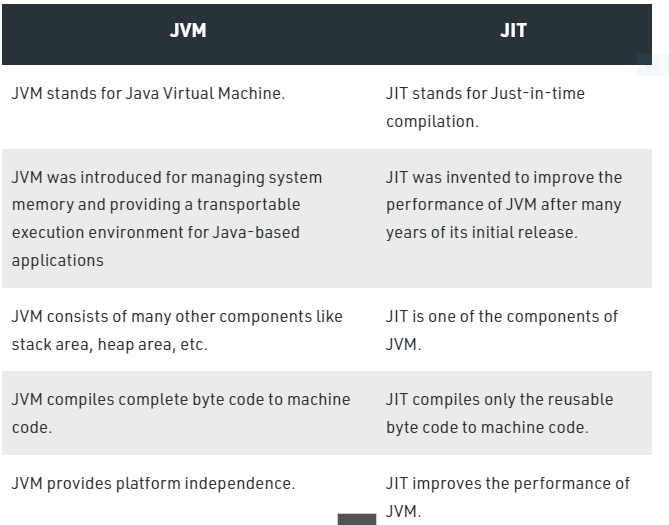
**Explain JVM architecture**

**explain class loader sub system**

**explain various runtime data areas**

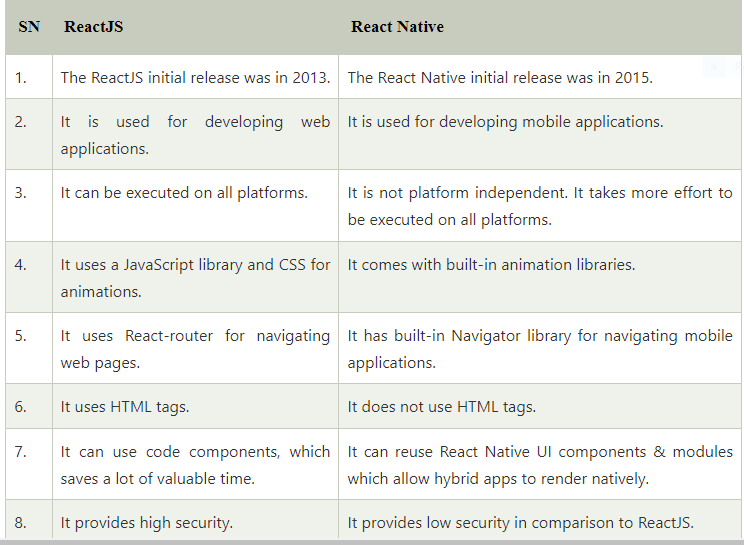
**explain execution engine**

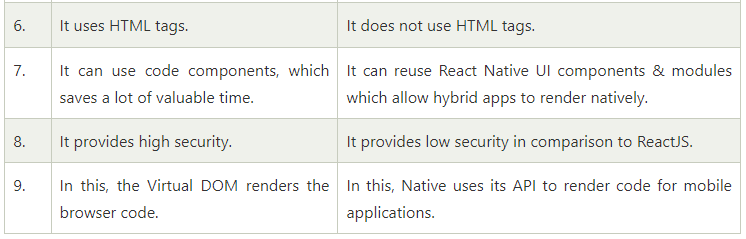
**javac compiler vs JIT compiler**



**what are the rules for overriding in java?**

**Overriding and Access-Modifiers :** The [access modifier](https://www.geeksforgeeks.org/access-modifiers-java/) for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.

**s**



**Exception Handling** in Java is one of the effective means to handle the runtime errors so that the regular flow of the application can be preserved. Java Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

**Exception** is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program’s instructions. Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

### ****Major reasons why an exception Occurs****

* Invalid user input
* Device failure
* Loss of network connection
* Physical limitations (out of disk memory)
* Code errors
* Opening an unavailable file

**Errors** represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

Let us discuss the most important part which is the **differences between Error and Exception** that is as follows:

* **Error:**An Error indicates a serious problem that a reasonable application should not try to catch.
* **Exception:**Exception indicates conditions that a reasonable application might try to catch.

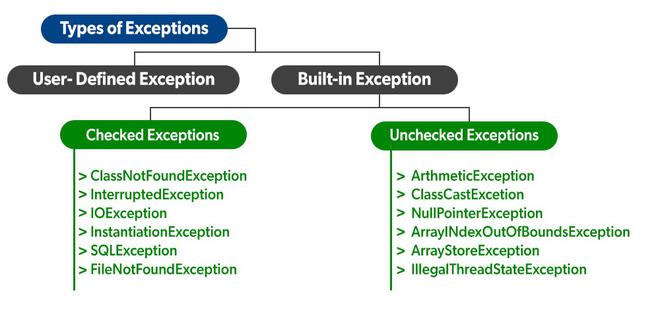
### Exception Hierarchy

All exception and error types are subclasses of class **Throwable**, which is the base class of the hierarchy. One branch is headed by **Exception**. This class is used for exceptional conditions that user programs should catch. NullPointerException is an example of such an exception. Another branch, **Error** is used by the Java run-time system([JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)) to indicate errors having to do with the run-time environment itself(JRE). StackOverflowError is an example of such an error.



### Types of Exceptions

Java defines several types of exceptions that relate to its various class libraries. Java also allows users to define their own exceptions.



**Exceptions can be categorized in two ways:**

1. **Built-in Exceptions**
   * Checked Exception
   * Unchecked Exception
2. **User-Defined Exceptions**

Let us discuss the above-defined listed exception that is as follows:

### ****A. Built-in Exceptions:****

Built-in exceptions are the exceptions that are available in Java libraries. These exceptions are suitable to explain certain error situations.

* **Checked Exceptions:**Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
* **Unchecked Exceptions:**The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error.

***Note:****For checked vs unchecked exception, see*[*Checked vs Unchecked Exceptions*](https://www.geeksforgeeks.org/checked-vs-unchecked-exceptions-in-java/)

### ****B. User-Defined Exceptions:****

Sometimes, the built-in exceptions in Java are not able to describe a certain situation. In such cases, users can also create exceptions, which are called ‘user-defined Exceptions’.

The ***advantages of Exception Handling in Java***are as follows:

1. Provision to Complete Program Execution
2. Easy Identification of Program Code and Error-Handling Code
3. Propagation of Errors
4. Meaningful Error Reporting
5. Identifying Error Types

**Methods to print the Exception information:**

**1.printStackTrace()**– This method prints exception information in the format of Name of the exception: description of the exception, stack

trace.

* Java

|  |
| --- |
| //program to print the exception information using printStackTrace() method    import java.io.\*;    class GFG {      public static void main (String[] args) {        int a=5;        int b=0;          try{            System.out.println(a/b);          }        catch(ArithmeticException e){          e.printStackTrace();        }      }  } |

**Output:**

java.lang.ArithmeticException: / by zero

at GFG.main(File.java:10)

**2.toString() – This** method prints exception information in the format of Name of the exception: description of the exception.

* Java

|  |
| --- |
| //program to print the exception information using toString() method    import java.io.\*;    class GFG1 {      public static void main (String[] args) {        int a=5;        int b=0;          try{            System.out.println(a/b);          }        catch(ArithmeticException e){          System.out.println(e.toString());        }      }  } |

**Output:**

java.lang.ArithmeticException: / by zero

**3.getMessage()** -This method prints only the description of the exception.

* Java

|  |
| --- |
| //program to print the exception information using getMessage() method    import java.io.\*;    class GFG1 {      public static void main (String[] args) {        int a=5;        int b=0;          try{            System.out.println(a/b);          }        catch(ArithmeticException e){          System.out.println(e.getMessage());        }      }  } |

**Output:**

/ by zero

### How Does JVM handle an Exception?

**Default Exception Handling:**Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred. Creating the Exception Object and handling it in the run-time system is called throwing an Exception. There might be a list of the methods that had been called to get to the method where an exception occurred. This ordered list of the methods is called **Call Stack**. Now the following procedure will happen.

* The run-time system searches the call stack to find the method that contains a block of code that can handle the occurred exception. The block of the code is called an **Exception handler**.
* The run-time system starts searching from the method in which the exception occurred, and proceeds through the call stack in the reverse order in which methods were called.
* If it finds an appropriate handler, then it passes the occurred exception to it. An appropriate handler means the type of the exception object thrown matches the type of the exception object it can handle.
* If the run-time system searches all the methods on the call stack and couldn’t have found the appropriate handler, then the run-time system handover the Exception Object to the **default exception handler**, which is part of the run-time system. This handler prints the exception information in the following format and terminates the program **abnormally**.

1. **int** a=50/0;//ArithmeticException
2. String s=**null**;
3. System.out.println(s.length());//NullPointerException

### 3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a [string](https://www.javatpoint.com/java-string) variable that has characters; converting this variable into digit will cause NumberFormatException.

1. String s="abc";
2. **int** i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

1. **int** a[]=**new** **int**[5];
2. a[10]=50; //ArrayIndexOutOfBoundsException
3. **InputMismatchException**.
4. System.out.println("Enter value of a to get its square value:");
5. Integer a = sc.nextInt(); // we give any float value as input
6. System.out.println((a\*a));
7. -----------------------------------------------------------------------------------

## Exceptions in Main Thread

In this section, we will see some common main thread exceptions that may occur in different scenarios. They are as follows:

**1. Exception in thread main java.lang.UnsupportedClassVersionError:** This exception occurs in a program when a java class is compiled from another JDK version and we are trying to run it from another java version.

The UnsupportedClassVersionError is present java.lang package.

**2. Exception in thread main java.lang.NoClassDefFoundError:** This exception occurs in two flavors. The first scenario is where we provide class full name with .class extension. The second scenario comes when Class is not found.

**3. Exception in thread main java.lang.NoSuchMethodError: main:** This exception occurs in a Java program when a class is trying to run without the main method declaration.

**4. Exception in thread main java.lang.ArithmeticException:** When any exception is thrown from main method, it prints the exception in the console.

The first part explains that exception is thrown from the main method, second part prints the exception class name and then after a colon, it prints an

# Why java.lang.VerifyError Occurs in Java and How to Solve this?

The Java Virtual Machine (JVM) distrusts all loaded bytecode as a core tenet of the Java Security Model. During runtime, the JVM will load .class files and attempt to link them together to form an executable — but the validity of these loaded .class files is unknown.  To ensure that the loaded .class files do not pose a threat to the final executable, a verification process is done on the .class files by the JVM.

Additionally, the JVM ensures that binaries are well-formed. For example, the JVM will verify classes do not subtype final classes. In many cases, verification fails on valid, non-malicious bytecode because a newer version of Java has a stricter verification process than older versions. For example, JDK 13 may have added a verification step that was not enforced in JDK 7. Thus, if we run an application with JVM 13 and include dependencies compiled with an older version of the Java Compiler (javac), the JVM may consider the outdated dependencies to be invalid.

Thus, when linking older .class files with a newer JVM, the JVM may throw a java.lang.VerifyError.

The VerifyError exists since the 1.0 version of Java.

**The Structure of VerifyError:**

**Constructors**

VerifyError()

This constructor creates an instance of the VerifyError class, setting null as its message.

VerifyError(String s)

This constructor creates an instance of the VerifyError class, using the specified string as message.  Here the class which threw the error is indicated through string argument.

**The three most common reasons upon which this error may occur as follows:**

**Reason 1:**“This error will be thrown whenever a class which is declared as final is extended.”

**Program:**

* Java

|  |
| --- |
| // Java program to show the occurrence  // of  java.lang.VerifyError  class B extends A {      public static void main(String args[])      {            System.out.println("my super class name:-"                             + myname);      }  }    public class A    {        static String myname = "A";  } |

As you see if you compile these two programs and execute it, it must have to work fine without showing any error. Now after changing the class A as follows and compile it alone.

final public class A

{

static String myname="A";

}

***Note****that here we have recompiled the “class A” alone. Now if we execute the class B (class that contains main() method) then an error message like below will be thrown at run-time.*

Exception in thread "main" java.lang.VerifyError: Cannot inherit from final class

at java.lang.ClassLoader.defineClass1(Native Method)

at java.lang.ClassLoader.defineClassCond(Unknown Source)

at java.lang.ClassLoader.defineClass(Unknown Source)

at java.security.SecureClassLoader.defineClass(Unknown Source)

at java.net.URLClassLoader.defineClass(Unknown Source)

at java.net.URLClassLoader.access$000(Unknown Source)

at java.net.URLClassLoader$1.run(Unknown Source)

at java.security.AccessController.doPrivileged(Native Method)

at java.net.URLClassLoader.findClass(Unknown Source)

at java.lang.ClassLoader.loadClass(Unknown Source)

at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)

at java.lang.ClassLoader.loadClass(Unknown Source)

Could not find the main class: B.

This error was caused because that we changed the definition of class TestClassA, but class TestClassB was compiled using an older version of the class TestClassA.

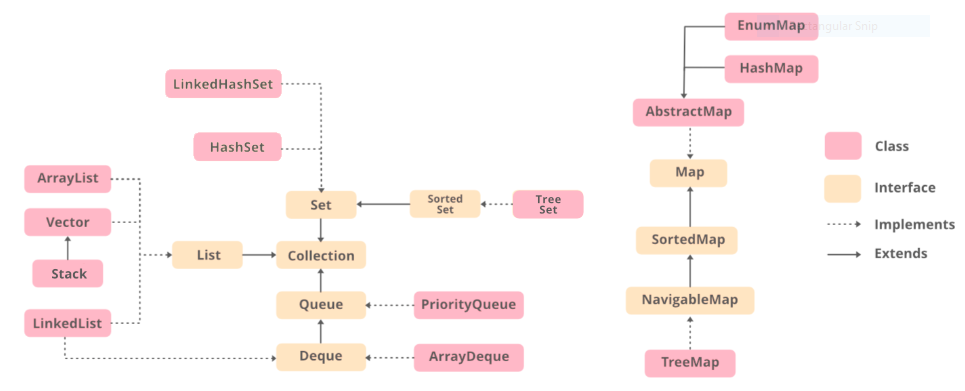
**Reason 2:**“Consider a class that extends another class before and if it no longer extends that class now, then this error may be thrown at run-time.”

**Program:**

**How to deal with the VerifyError?**

In order to avoid the VerifyError, you must compile all your classes using the same version of Java. Also, once a change is done to a class, then make sure that you re-compile your project from scratch. Finally, if your application makes use of external libraries, verify that you use the appropriate version of every library and of course, consult the corresponding javadocs, in order to be sure that everything is correct.

Whenever possible, use the latest versions of dependencies rather than disabling verification.



| Method Overloading | Method Overriding |
| --- | --- |
| Method overloading is a compile-time polymorphism. | Method overriding is a run-time polymorphism. |
| It helps to increase the readability of the program. | It is used to grant the specific implementation of the method which is already provided by its parent class or superclass. |
| It occurs within the class. | It is performed in two classes with inheritance relationships. |
| Method overloading may or may not require inheritance. | Method overriding always needs inheritance. |
| In method overloading, methods must have the same name and different signatures. | In method overriding, methods must have the same name and same signature. |
| In method overloading, the return type can or can not be the same, but we just have to change the parameter. | In method overriding, the return type must be the same or co-variant. |
| Static binding is being used for overloaded methods. | Dynamic binding is being used for overriding methods. |
| It gives better performance. The reason behind this is that the binding of overridden methods is being done at runtime. | Poor performance |
| Private and final methods can be overloaded. | Private and final methods can’t be overridden. |
| Argument list should be different while doing method overloading. | Argument list should be same in method overriding. |

he default version of the clone method creates a shallow copy of an object. In java being object-oriented, object copying is creating a copy of the existing object, so the object copied can be a similar copy of the exact copy of the object of which it is copied. There are 2 ways to copy an object as follows:

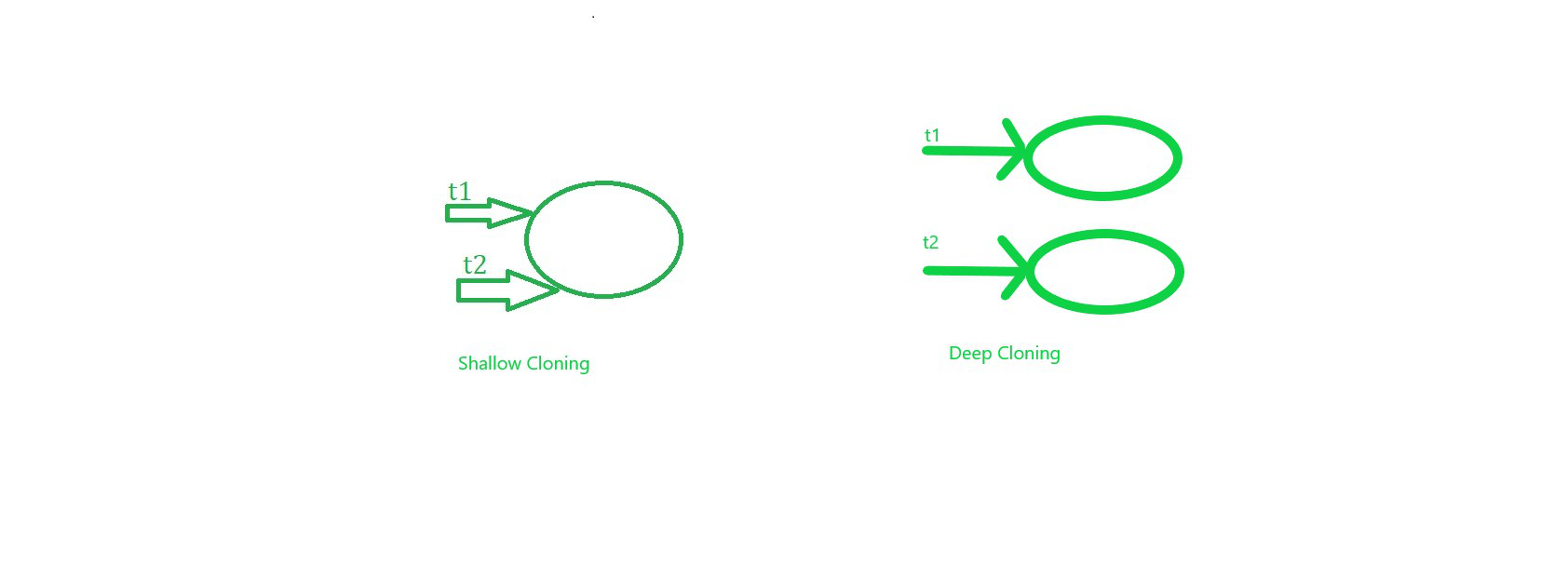
In C/C++ there

**Methods of Copying**

Now, copying can be of three types as follows:

1. Shallow copy
2. Deep copy
3. Cloning

Generally[,](https://www.geeksforgeeks.org/clone-method-in-java-2/) deep copy and cloning is referred to as the same deep cloning as the difference between them is a thin line where focus is laid to ease for copying objects manually using the [clone()](https://www.geeksforgeeks.org/clone-method-in-java-2/) method.



**1. Shallow Copy/ Cloning**

* Default implementation while using the clone() method a shallow copy of the object is created. It means it creates a new instance and copies all the fields of the object to that new instance where both are referencing to the same memory in heap memory.
* [*clone()*](https://www.geeksforgeeks.org/clone-method-in-java-2/) method of the object class supports the shallow copy of the object. If the object contains primitive as well as non-primitive or reference type variable in shallow copy, the cloned object also refers to the same object to which the original object refers as only the object references get copied and not the referred objects themselves. In simpler terms, One object is being created here while two references in the stack memory.
* It returns it as an object type, we need to explicitly cast it back to our original object. This is a shallow copy of the object.

Illustration: Shallow copying object ‘t2’

class GFG

{

int i,j;

}

main(String[] args)

{

// **Copying**

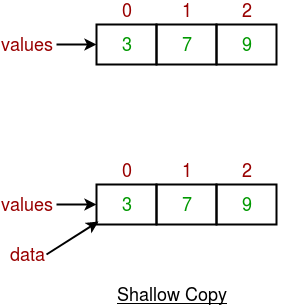
GFG object1 = new GFG() ;

object.i = 5;

object.j = 6;

**GFG object2 = object1 ;** // shallow copying object

**}**



**Example 1**

* Java

|  |
| --- |
| // Java Program to show shallow cloning    // Importing java input/output classes  import java.io.\*;    // Class  class GFG {        // Main driver method      public static void main(String[] args)      {          // t1 and t2 objects are used to          // illustrate shallow copy            // t1 is first object created in heap memory          GFG t1 = new GFG();            // Creating only one object(t1) and          // both objects (t1,t2) are pointing to only one          // object          GFG t2 = t1;          // Display message          // return true if reference is same that is shallow copy          // false if different          // Should be returning - true          System.out.print(                  "Output: False if Deepcopy & True if shallow : ");                  System.out.println(t1 == t2);      }  } |

**Output**

Output: False if Deepcopy & True if shallow : true

Output explanation:

1. As seen above there were two reference variables t1, t2. Whenever the object of GFG class is created the t1 (reference variable) is pointing to one object.
2. Afterward, assigning the t1 reference variable into the new reference variable t2 of the same class pointing to the same object in heap memory. Both t1 and t2 reference variables are pointing to only one object.
3. It does not create a new duplicate object. The above two statements prove that both t1 and t2 reference variables are point only one object.

**2. Deep copy/ cloning**is the process of creating exactly the independent duplicate objects in the heap memory and manually assigning the values of the second object where values are supposed to be copied is called deep cloning.

* Whenever there is a need self copy not using default implementation is referred to as deep copy where the object to be needed is implemented as per needs. Therefore, for the deep copy, it needs to ensure that all the member classes also implement the Cloneable interface because of overriding the clone() method of the object class.

GFG object1 = new GFG() ; || Creating object of GFG class

object.i = 5;

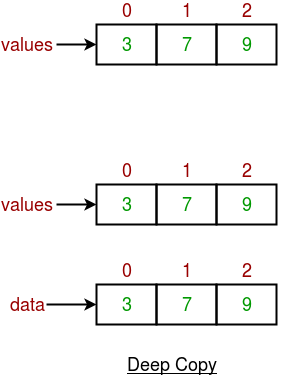
object.j = 6;

GFG object2 = new GFG; || Creating another object class

object2.i = object1.i ; || Deep copying

object2.j = object1.j ; || Deep copying

*Note: In can not directly call clone() method using object.clone() as clone method in Object class of java is protected. So override the function by making access modifier as public in return super.clone() from the class created to override*



**Example 2**

* Java

|  |
| --- |
| // Java Program to show Deep Cloning    // Importing java input/output libraries  import java.io.\*;    // Class  public class GFG implements Cloneable {      // Defining a method as clone method is protected    // Defining within the class called cloneable interface    public Object clone() throws CloneNotSupportedException    {    return (GFG)super.clone();    }        // Main driver method      public static void main(String[] args) throws CloneNotSupportedException      {          // Creating first object of GFG class          GFG t1 = new GFG();            // Using clone()  method to create duplicate object          // of t1 reference variable          // else for every object manually object needs          // to be copied in deep copying          // clone() ease this manual effort          GFG t2 = (GFG)t1.clone();            // Comparing two objects just after deep copying          // Returning true for shallow(by default) copying          // Returning false for deep copying          System.out.println(t1 == t2);      }  } |

**Output**

false

Output explanation:

1. Here, there are two reference variables t1, t2. When the object of GFG class is created, the t1 (reference variable) is pointing to one object.
2. Afterward, t1 is assigned as a reference variable and calls [*clone()*](https://www.geeksforgeeks.org/clone-method-in-java-2/) method into a new reference variable t2 of the same class. Both t1 and t2 reference variables are pointing to different objects. Here t2 has created a new duplicate object and t1 also.
3. The above two statements prove that both t1 and t2 reference variable are point difference object if t1 point to O1 object than t2 point to O2 object.

# Java 8 Features

Oracle released a new version of Java as Java 8 in March 18, 2014. It was a revolutionary release of the Java for software development platform. It includes various upgrades to the Java programming, JVM, Tools and libraries.

## Java 8 Programming Language Enhancements

Java 8 provides following features for Java Programming:

* Lambda expressions,
* Method references,
* Functional interfaces,
* Stream API,
* Default methods,
* Base64 Encode Decode,
* Static methods in interface,
* Optional class,
* Collectors class,
* ForEach() method,
* Nashorn JavaScript Engine,
* Parallel Array Sorting,
* Type and Repating Annotations,
* IO Enhancements,
* Concurrency Enhancements,
* JDBC Enhancements etc.

# Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

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.

46.5M

791

History of Java

**Next**

**Stay**

## Why use Lambda Expression

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

## Java Lambda Expression Syntax

1. (argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**No Parameter Syntax**

1. () -> {
2. //Body of no parameter lambda
3. }

**One Parameter Syntax**

1. (p1) -> {
2. //Body of single parameter lambda
3. }

**Two Parameter Syntax**

1. (p1,p2) -> {
2. //Body of multiple parameter lambda
3. }

Let's see a scenario where we are not implementing Java lambda expression. Here, we are implementing an interface without using lambda expression.

## Without Lambda Expression

1. **interface** Drawable{
2. **public** **void** draw();
3. }
4. **public** **class** LambdaExpressionExample {
5. **public** **static** **void** main(String[] args) {
6. **int** width=10;
8. //without lambda, Drawable implementation using anonymous class
9. Drawable d=**new** Drawable(){
10. **public** **void** draw(){System.out.println("Drawing "+width);}
11. };
12. d.draw();
13. }
14. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample" \t "_blank)**

Output:

Drawing 10

## Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

1. @FunctionalInterface  //It is optional
2. **interface** Drawable{
3. **public** **void** draw();
4. }
6. **public** **class** LambdaExpressionExample2 {
7. **public** **static** **void** main(String[] args) {
8. **int** width=10;
10. //with lambda
11. Drawable d2=()->{
12. System.out.println("Drawing "+width);
13. };
14. d2.draw();
15. }
16. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample2" \t "_blank)**

Output:

Drawing 10

A lambda expression can have zero or any number of arguments. Let's see the examples:

## Java Lambda Expression Example: No Parameter

1. **interface** Sayable{
2. **public** String say();
3. }
4. **public** **class** LambdaExpressionExample3{
5. **public** **static** **void** main(String[] args) {
6. Sayable s=()->{
7. **return** "I have nothing to say.";
8. };
9. System.out.println(s.say());
10. }
11. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample3" \t "_blank)**

Output:

I have nothing to say.

## Java Lambda Expression Example: Single Parameter

1. **interface** Sayable{
2. **public** String say(String name);
3. }
5. **public** **class** LambdaExpressionExample4{
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression with single parameter.
9. Sayable s1=(name)->{
10. **return** "Hello, "+name;
11. };
12. System.out.println(s1.say("Sonoo"));
14. // You can omit function parentheses
15. Sayable s2= name ->{
16. **return** "Hello, "+name;
17. };
18. System.out.println(s2.say("Sonoo"));
19. }
20. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample4" \t "_blank)**

Output:

Hello, Sonoo

Hello, Sonoo

## Java Lambda Expression Example: Multiple Parameters

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample5{
6. **public** **static** **void** main(String[] args) {
8. // Multiple parameters in lambda expression
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Multiple parameters with data type in lambda expression
13. Addable ad2=(**int** a,**int** b)->(a+b);
14. System.out.println(ad2.add(100,200));
15. }
16. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample5" \t "_blank)**

Output:

30

300

## Java Lambda Expression Example: with or without return keyword

In Java lambda expression, if there is only one statement, you may or may not use return keyword. You must use return keyword when lambda expression contains multiple statements.

1. **interface** Addable{
2. **int** add(**int** a,**int** b);
3. }
5. **public** **class** LambdaExpressionExample6 {
6. **public** **static** **void** main(String[] args) {
8. // Lambda expression without return keyword.
9. Addable ad1=(a,b)->(a+b);
10. System.out.println(ad1.add(10,20));
12. // Lambda expression with return keyword.
13. Addable ad2=(**int** a,**int** b)->{
14. **return** (a+b);
15. };
16. System.out.println(ad2.add(100,200));
17. }
18. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample6" \t "_blank)**

Output:

30

300

## Java Lambda Expression Example: Foreach Loop

1. **import** java.util.\*;
2. **public** **class** LambdaExpressionExample7{
3. **public** **static** **void** main(String[] args) {
5. List<String> list=**new** ArrayList<String>();
6. list.add("ankit");
7. list.add("mayank");
8. list.add("irfan");
9. list.add("jai");
11. list.forEach(
12. (n)->System.out.println(n)
13. );
14. }
15. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample7" \t "_blank)**

Output:

ankit

mayank

irfan

jai

## Java Lambda Expression Example: Multiple Statements

1. @FunctionalInterface
2. **interface** Sayable{
3. String say(String message);
4. }
6. **public** **class** LambdaExpressionExample8{
7. **public** **static** **void** main(String[] args) {
9. // You can pass multiple statements in lambda expression
10. Sayable person = (message)-> {
11. String str1 = "I would like to say, ";
12. String str2 = str1 + message;
13. **return** str2;
14. };
15. System.out.println(person.say("time is precious."));
16. }
17. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample8" \t "_blank)**

Output:

I would like to say, time is precious.

## Java Lambda Expression Example: Creating Thread

You can use lambda expression to run thread. In the following example, we are implementing run method by using lambda expression.

1. **public** **class** LambdaExpressionExample9{
2. **public** **static** **void** main(String[] args) {
4. //Thread Example without lambda
5. Runnable r1=**new** Runnable(){
6. **public** **void** run(){
7. System.out.println("Thread1 is running...");
8. }
9. };
10. Thread t1=**new** Thread(r1);
11. t1.start();
12. //Thread Example with lambda
13. Runnable r2=()->{
14. System.out.println("Thread2 is running...");
15. };
16. Thread t2=**new** Thread(r2);
17. t2.start();
18. }
19. }

**[Test it Now](https://compiler.javatpoint.com/opr/test.jsp?filename=LambdaExpressionExample9" \t "_blank)**

Output:

Thread1 is running...

Thread2 is running...

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

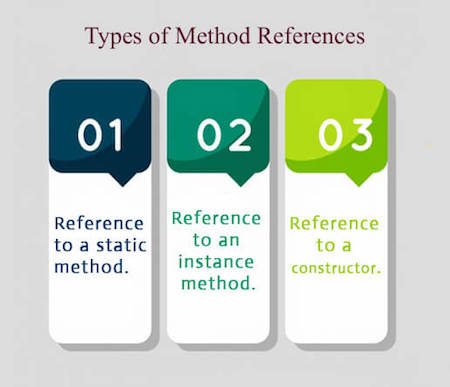
# Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

## Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.



## 1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

63M

1.3K

OOPs Concepts in Java

1. ContainingClass::staticMethodName

### Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** MethodReference {
5. **public** **static** **void** saySomething(){
6. System.out.println("Hello, this is static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. // Referring static method
10. Sayable sayable = MethodReference::saySomething;
11. // Calling interface method
12. sayable.say();
13. }
14. }

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

Output:

Hello, this is static method.

### Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

1. **public** **class** MethodReference2 {
2. **public** **static** **void** ThreadStatus(){
3. System.out.println("Thread is running...");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(MethodReference2::ThreadStatus);
7. t2.start();
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference2)

Output:

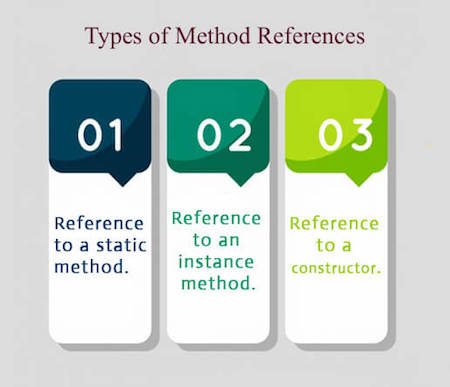
# Java Method References

Java provides a new feature called method reference in Java 8. Method reference is used to refer method of functional interface. It is compact and easy form of lambda expression. Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference. In this tutorial, we are explaining method reference concept in detail.

## Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.



## 1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

1. ContainingClass::staticMethodName

### Example 1

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

1. **interface** Sayable{
2. **void** say();
3. }
4. **public** **class** MethodReference {
5. **public** **static** **void** saySomething(){
6. System.out.println("Hello, this is static method.");
7. }
8. **public** **static** **void** main(String[] args) {
9. // Referring static method
10. Sayable sayable = MethodReference::saySomething;
11. // Calling interface method
12. sayable.say();
13. }
14. }

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

Output:

Hello, this is static method.

### Example 2

In the following example, we are using predefined functional interface Runnable to refer static method.

1. **public** **class** MethodReference2 {
2. **public** **static** **void** ThreadStatus(){
3. System.out.println("Thread is running...");
4. }
5. **public** **static** **void** main(String[] args) {
6. Thread t2=**new** Thread(MethodReference2::ThreadStatus);
7. t2.start();
8. }
9. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference2)

Output:

Thread is running...

### Example 3

You can also use predefined functional interface to refer methods. In the following example, we are using BiFunction interface and using it's apply() method.

1. **import** java.util.function.BiFunction;
2. **class** Arithmetic{
3. **public** **static** **int** add(**int** a, **int** b){
4. **return** a+b;
5. }
6. }
7. **public** **class** MethodReference3 {
8. **public** **static** **void** main(String[] args) {
9. BiFunction<Integer, Integer, Integer>adder = Arithmetic::add;
10. **int** result = adder.apply(10, 20);
11. System.out.println(result);
12. }
13. }

[**Test it Now**](https://compiler.javatpoint.com/opr/test.jsp?filename=MethodReference3)

Output:

30

# Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

### Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

1. **interface** Sayable{
2. // Default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. }
9. **public** **class** DefaultMethods **implements** Sayable{
10. **public** **void** sayMore(String msg){        // implementing abstract method
11. System.out.println(msg);
12. }
13. **public** **static** **void** main(String[] args) {
14. DefaultMethods dm = **new** DefaultMethods();
15. dm.say();   // calling default method
16. dm.sayMore("Work is worship");  // calling abstract method
18. }
19. }

Output:

Play Videox

Hello, this is default method

Work is worship

## Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

1. **interface** Sayable{
2. // default method
3. **default** **void** say(){
4. System.out.println("Hello, this is default method");
5. }
6. // Abstract method
7. **void** sayMore(String msg);
8. // static method
9. **static** **void** sayLouder(String msg){
10. System.out.println(msg);
11. }
12. }
13. **public** **class** DefaultMethods **implements** Sayable{
14. **public** **void** sayMore(String msg){     // implementing abstract method
15. System.out.println(msg);
16. }
17. **public** **static** **void** main(String[] args) {
18. DefaultMethods dm = **new** DefaultMethods();
19. dm.say();                       // calling default method
20. dm.sayMore("Work is worship");      // calling abstract method
21. Sayable.sayLouder("Helloooo...");   // calling static method
22. }
23. }

Output:

Hello there

Work is worship

Helloooo...

## Abstract Class vs Java 8 Interface

After having default and static methods inside the interface, we think about the need of abstract class in Java. An interface and an abstract class is almost similar except that you can create constructor in the abstract class whereas you can't do this in interface.

1. **abstract** **class** AbstractClass{
2. **public** AbstractClass() {        // constructor
3. System.out.println("You can create constructor in abstract class");
4. }
5. **abstract** **int** add(**int** a, **int** b); // abstract method
6. **int** sub(**int** a, **int** b){      // non-abstract method
7. **return** a-b;
8. }
9. **static** **int** multiply(**int** a, **int** b){  // static method
10. **return** a\*b;
11. }
12. }
13. **public** **class** AbstractTest **extends** AbstractClass{
14. **public** **int** add(**int** a, **int** b){        // implementing abstract method
15. **return** a+b;
16. }
17. **public** **static** **void** main(String[] args) {
18. AbstractTest a = **new** AbstractTest();
19. **int** result1 = a.add(20, 10);    // calling abstract method
20. **int** result2 = a.sub(20, 10);    // calling non-abstract method
21. **int** result3 = AbstractClass.multiply(20, 10); // calling static method
22. System.out.println("Addition: "+result1);
23. System.out.println("Substraction: "+result2);
24. System.out.println("Multiplication: "+result3);
25. }
26. }

Output:

You can create constructor in abstract class

Addition: 30

Substraction: 10

Multiplication:

# Java forEach loop

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

## forEach() Signature in Iterable Interface

1. **default** **void** forEach(Consumer<**super** T>action)

### Java 8 forEach() example 1

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression--------------");
11. gamesList.forEach(games -> System.out.println(games));
13. }
14. }

Output:

------------Iterating by passing lambda expression--------------

Football

Cricket

Chess

Hocky

### Java 8 forEach() example 2

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing method reference---------------");
11. gamesList.forEach(System.out::println);
12. }
13. }

Output:

49.3M

880

Prime Ministers of India | List of Prime Minister of India (1947-2020)

------------Iterating by passing method reference---------------

Football

Cricket

Chess

Hocky

## Java Stream forEachOrdered() Method

Along with forEach() method, Java provides one more method forEachOrdered(). It is used to iterate elements in the order specified by the stream.

### Singnature:

1. **void** forEachOrdered(Consumer<? **super** T> action)

### Java Stream forEachOrdered() Method Example

1. **import** java.util.ArrayList;
2. **import** java.util.List;
3. **public** **class** ForEachOrderedExample {
4. **public** **static** **void** main(String[] args) {
5. List<String> gamesList = **new** ArrayList<String>();
6. gamesList.add("Football");
7. gamesList.add("Cricket");
8. gamesList.add("Chess");
9. gamesList.add("Hocky");
10. System.out.println("------------Iterating by passing lambda expression---------------");
11. gamesList.stream().forEachOrdered(games -> System.out.println(games));
12. System.out.println("------------Iterating by passing method reference---------------");
13. gamesList.stream().forEachOrdered(System.out::println);
14. }
16. }

Output:

------------Iterating by passing lambda expression---------------

Football

Cricket

Chess

Hocky

------------Iterating by passing method reference---------------

Football

Cricket

Chess

Hocky

# Java Stream Filter

Java stream provides a method filter() to filter stream elements on the basis of given predicate. Suppose you want to get only even elements of your list then you can do this easily with the help of filter method.

This method takes predicate as an argument and returns a stream of consisting of resulted elements.

## Signature

The signature of Stream filter() method is given below:

1. Stream<T> filter(Predicate<? **super** T> predicate)

### Parameter

**predicate:** It takes Predicate reference as an argument. Predicate is a functional interface. So, you can also pass lambda expression here.

Play Videox

### Return

It returns a new stream.

### Java Stream filter() example

In the following example, we are fetching and iterating filtered data.

1. **import** java.util.\*;
2. **class** Product{
3. **int** id;
4. String name;
5. **float** price;
6. **public** Product(**int** id, String name, **float** price) {
7. **this**.id = id;
8. **this**.name = name;
9. **this**.price = price;
10. }
11. }
12. **public** **class** JavaStreamExample {
13. **public** **static** **void** main(String[] args) {
14. List<Product> productsList = **new** ArrayList<Product>();
15. //Adding Products
16. productsList.add(**new** Product(1,"HP Laptop",25000f));
17. productsList.add(**new** Product(2,"Dell Laptop",30000f));
18. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
19. productsList.add(**new** Product(4,"Sony Laptop",28000f));
20. productsList.add(**new** Product(5,"Apple Laptop",90000f));
21. productsList.stream()
22. .filter(p ->p.price> 30000)   // filtering price
23. .map(pm ->pm.price)          // fetching price
24. .forEach(System.out::println);  // iterating price
25. }
26. }

Output:

90000.0

### Java Stream filter() example 2

In the following example, we are fetching filtered data as a list.

1. **import** java.util.\*;
2. **import** java.util.stream.Collectors;
3. **class** Product{
4. **int** id;
5. String name;
6. **float** price;
7. **public** Product(**int** id, String name, **float** price) {
8. **this**.id = id;
9. **this**.name = name;
10. **this**.price = price;
11. }
12. }
13. **public** **class** JavaStreamExample {
14. **public** **static** **void** main(String[] args) {
15. List<Product> productsList = **new** ArrayList<Product>();
16. //Adding Products
17. productsList.add(**new** Product(1,"HP Laptop",25000f));
18. productsList.add(**new** Product(2,"Dell Laptop",30000f));
19. productsList.add(**new** Product(3,"Lenevo Laptop",28000f));
20. productsList.add(**new** Product(4,"Sony Laptop",28000f));
21. productsList.add(**new** Product(5,"Apple Laptop",90000f));
22. List<Float> pricesList =  productsList.stream()
23. .filter(p ->p.price> 30000)   // filtering price
24. .map(pm ->pm.price)          // fetching price
25. .collect(Collectors.toList());
26. System.out.println(pricesList);
27. }
28. }

[Iterator](https://www.geeksforgeeks.org/iterators-in-java/)

[Iterators](https://www.geeksforgeeks.org/iterators-in-java/) are used in [Collection framework in Java](https://www.geeksforgeeks.org/collections-in-java-2/) to retrieve elements one by one. It can be applied to any Collection object. By using Iterator, we can perform both read and remove operations. Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like [Set](https://www.geeksforgeeks.org/set-in-java/), [List](https://www.geeksforgeeks.org/list-interface-java-examples/), [Queue](https://www.geeksforgeeks.org/queue-interface-java/), [Deque](https://www.geeksforgeeks.org/deque-interface-java-example/" \t "_blank) and also in all implemented classes of [Map interface](https://www.geeksforgeeks.org/map-interface-java-examples/). Iterator is the only cursor available for entire collection framework.

Iterator object can be created by calling iterator() method present in Collection interface.

// Here "c" is any Collection object. itr is of

// type Iterator interface and refers to "c"

Iterator itr = c.iterator();

[**ListIterator**](https://www.geeksforgeeks.org/iterators-in-java/)  
It is only applicable for [List collection](https://www.geeksforgeeks.org/list-interface-java-examples/) implemented classes like [arraylist](https://www.geeksforgeeks.org/arraylist-in-java/), [linkedlist](https://www.geeksforgeeks.org/linked-list-in-java/" \t "_blank) etc. It provides bi-directional iteration. ListIterator must be used when we want to enumerate elements of List. This cursor has more functionality(methods) than iterator.

ListIterator object can be created by calling listIterator() method present in List interface.

// Here "l" is any List object, ltr is of type

// ListIterator interface and refers to "l"

ListIterator ltr = l.listIterator();

**Differences between Iterator and ListIterator:**

1. Iterator can traverse only in forward direction whereas ListIterator traverses both in forward and backward directions.

**Example:**

|  |
| --- |
| import java.io.\*;  import java.util.\*;    class IteratorDemo1 {      public static void main(String[] args)      {          ArrayList<Integer> list              = new ArrayList<Integer>();            list.add(1);          list.add(2);          list.add(3);          list.add(4);          list.add(5);            // Iterator          Iterator itr = list.iterator();            System.out.println("Iterator:");          System.out.println("Forward traversal: ");            while (itr.hasNext())              System.out.print(itr.next() + " ");            System.out.println();            // ListIterator          ListIterator i = list.listIterator();            System.out.println("ListIterator:");          System.out.println("Forward Traversal : ");            while (i.hasNext())              System.out.print(i.next() + " ");            System.out.println();            System.out.println("Backward Traversal : ");            while (i.hasPrevious())              System.out.print(i.previous() + " ");            System.out.println();      }  } |

**Output:**

Iterator:

Forward traversal:

1 2 3 4 5

ListIterator:

Forward Traversal :

1 2 3 4 5

Backward Traversal :

5 4 3 2 1

1. ListIterator can help to replace an element whereas Iterator cannot.

**Example:**

|  |
| --- |
| import java.util.ArrayList;  import java.util.ListIterator;    public class ListIteratorDemo2 {      public static void main(String[] args)      {            ArrayList<Integer> aList              = new ArrayList<Integer>();          aList.add(1);          aList.add(2);          aList.add(3);          aList.add(4);          aList.add(5);            System.out.println("Elements of ArrayList: ");          for (Integer i : aList) {              System.out.println(i);          }          ListIterator<Integer> l              = aList.listIterator();          l.next();          l.set(80000);            System.out.println("\nNow the ArrayList"                             + " elements are: ");          for (Integer i : aList) {              System.out.println(i);          }      }  } |

**Output:**

Elements of ArrayList:

1

2

3

4

5

Now the ArrayList elements are:

80000

2

3

4

5

**OUTPUT**

**Table showing Difference between Iterator and ListIterator**

| Iterator | ListIterator |
| --- | --- |
| Can traverse elements present in Collection only in the forward direction. | Can traverse elements present in Collection both in forward and backward directions. |
| Helps to traverse Map, List and Set. | Can only traverse List and not the other two. |
| Indexes cannot be obtained by using Iterator. | It has methods like nextIndex() and previousIndex() to obtain indexes of elements at any time while traversing List. |
| Cannot modify or replace elements present in Collection | We can modify or replace elements with the help of set(E e) |
| Cannot add elements and it throws ConcurrentModificationException. | Can easily add elements to a collection at any time. |
| Certain methods of Iterator are next(), remove() and hasNext(). | Certain methods of ListIterator are next(), previous(), hasNext(), hasPrevious(), add(E e). |

# Difference between Iterator and Enumeration in Java with Examples

* Last Updated : 28 Jan, 2021

 Read

 Discuss

[**Iterator**](https://www.geeksforgeeks.org/iterators-in-java/)**:**It is a universal iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is an improved version of Enumeration with the additional functionality of remove-ability of an element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the only cursor available for entire collection framework.

**Syntax:** 

// Here "c" is any Collection object. itr is of

// type Iterator interface and refers to "c"

Iterator itr = c.iterator();

[**Enumeration**](https://www.geeksforgeeks.org/enum-in-java/)**:**Enumeration (or enum) is a user-defined data type. It is mainly used to assign names to integral constants, the names make a program easy to read and maintain. In Java (from 1.5), enums are represented using the enum data type. Java enums are more powerful than C/C++ enums. In Java, we can also add variables, methods and constructors to it. The main objective of the enum is to define our own data types(Enumerated Data Types).

**Syntax:** 

// A simple enum example where enum is declared

// outside any class (Note enum keyword instead of

// class keyword)

enum Color

{

RED, GREEN, BLUE;

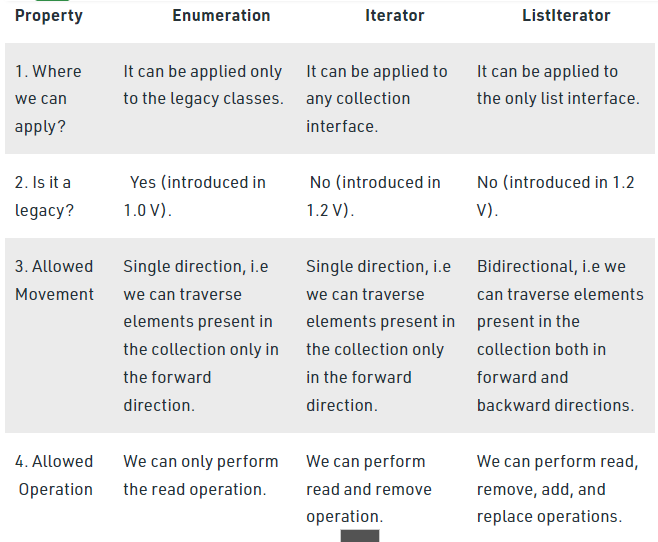
}

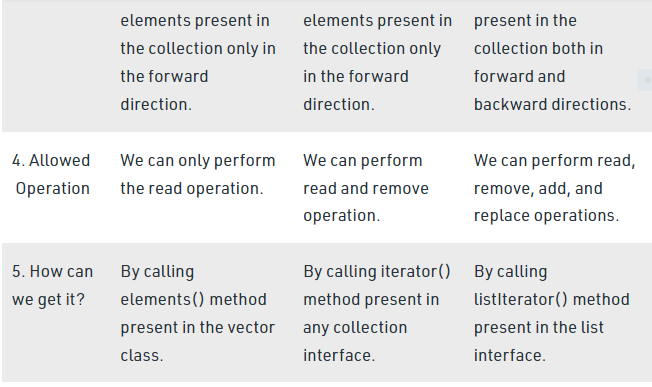
**Difference between Iterator and Enumeration:**

The functionality of Enumeration and the Iterator are same. Using Enumeration you can only traverse and fetch the objects, where as using Iterator we can also add and remove the objects. So Iterator can be useful if you want   
to manipulate the list and Enumeration is for read-only access. 

| Iterator | Enumeration |
| --- | --- |
| Iterator is a universal cursor as it is applicable for all the collection classes. | Enumeration is not a universal cursor as it applies only to legacy classes. |
| Iterator has the remove() method. | Enumeration does not have the remove() method. |
| Iterator can do modifications (e.g using remove() method it removes the element from the Collection during traversal). | Enumeration interface acts as a read only interface, one can not do any modifications to Collection while traversing the elements of the Collection. |
| Iterator is not a legacy interface. Iterator can be used for the traversal of HashMap, LinkedList, ArrayList, HashSet, TreeMap, TreeSet . |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Property** | **Enumeration** | **Iterator** | **ListIterator** |
| 1. Where we can apply? | It can be applied only to the legacy classes. | It can be applied to any collection interface. | It can be applied to the only list interface. |
| 2. Is it a legacy? | Yes (introduced in 1.0 V). | No (introduced in 1.2 V). | No (introduced in 1.2 V). |
| 3. Allowed Movement | Single direction, i.e we can traverse elements present in the collection only in the forward direction. | Single direction, i.e we can traverse elements present in the collection only in the forward direction. | Bidirectional, i.e we can traverse elements present in the collection both in forward and backward directions. |
| 4. Allowed  Operation | We can only perform the read operation. | We can perform read and remove operation. | We can perform read, remove, add, and replace operations. |
| 5. How can we get it? | By calling elements() method present in the vector class. | By calling iterator() method present in any collection interface. | By calling listIterator() method present in the list interface. |





# How to Avoid ConcurrentModificationException in Java?

* Difficulty Level : [Medium](https://www.geeksforgeeks.org/medium/)
* Last Updated : 16 Jun, 2021

 Read

 Discuss

[ConcurrentModificationException](https://www.geeksforgeeks.org/concurrentmodificationexception-in-java-with-examples/) is a predefined Exception in Java, which occurs while we are using Java Collections,  i.e whenever we try to modify an object concurrently without permission  ConcurrentModificationException occurs which is present in*java.util*package.

**Procedure:**Some steps are required in order to avoid the occurrence of this exception in a single-threaded environment. They are as follows:

* Instead of iterating through collections classes**,** we can iterate through the arrays. This works perfectly fine for smaller lists what about the bigger list? It’s very basic we know that if the array size is huge then it affects the performance. Hence, this method is only effective for smaller size arrays.
* The next method is the **Synchronized block method**, Here we actually lock the list in a synchronized block to avoid the exception. Isn’t that cool? but guess what this is also not an effective method to avoid Exception Why? Because the purpose of multithreading is not being used.
* The better way is we have [ConcurrentHashMap](https://www.geeksforgeeks.org/concurrenthashmap-in-java/) and CopyOnWriteArrayList Which is the best among the above Methods.

**Methods:**

Here two ways are proposed of which starting with the naive one and ending up with the optimal approach to reach the goal.

1. **Using Loops:** We used the Iterator remove() method instead of that we can use a for loop to avoid ConcurrentModificationException in a Single-threaded environment. If we add any extra objects then we can ensure that our code takes care of them.
2. **Using the remove() Method:**We can use the remove method to remove the object from the collection. Here there is a problem that is you can  only remove the same object and not any other from the list

**Example 1:**

* Java

|  |
| --- |
| // Java Program to Avoid  // AvoidConcurrentModificationException    // Importing Map and List utility classes  //  from java.util package  import java.util.List;  import java.util.Map;    // Importing classes from java.util.concurrent package  import java.util.concurrent.ConcurrentHashMap;  import java.util.concurrent.CopyOnWriteArrayList;    // Main class  public class GFG {        // Main driver method      public static void main(String[] args)      {            // Creating an object of List class          // Declaring object of string type          List<String> marvel              = new CopyOnWriteArrayList<String>();            // Adding elements to the above object created          // Custom input entries          marvel.add("IronMan");          marvel.add("BlackWidow");          marvel.add("Hulk");          marvel.add("DoctorStrange");          marvel.add("SpiderMan");            // Iterating over object created using size() method          for (int i = 0; i < marvel.size(); i++) {                // Print and display the object elements              // using get() method              System.out.println("Avenger : "                                 + marvel.get(i));                // Condition check over object elements                // If specific element is matched              if (marvel.get(i).equals("BlackWidow")) {                    marvel.remove(i);                  i--;                    // Add this specific element                  marvel.add("CaptianAmerica");              }          }            // Now getting the final total size by checking          // how many elements are there inside object          System.out.println("Total Avengers : "                             + marvel.size());      }  } |

**Output**

Avenger : IronMan

Avenger : BlackWidow

Avenger : Hulk

Avenger : DoctorStrange

Avenger : SpiderMan

Avenger : CaptianAmerica

Total Avengers :5

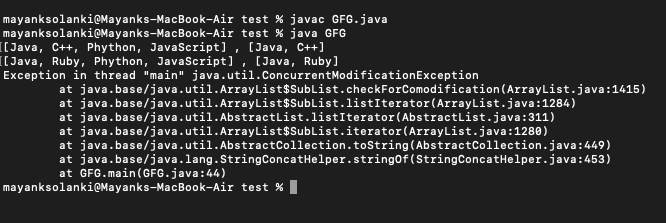
***Note:****The Exception can also occur if we try to modify the structure of original list with sublist. An example for the same is below,*

**Example 2:**

* Java

|  |
| --- |
| // Java Program to Illustrate ConcurrentModificationException  // WithArrayListSubList    // Importing List and Arraylist classes utility classes  // from java.util package  import java.util.ArrayList;  import java.util.List;    // Main class  public class GFG {        // Main driver method      public static void main(String[] args) {            // Creating a List class object          // Declaring object of string type          List <String> names = new ArrayList <>();            // Adding elements to the object of List class            // Custom input entries          names.add("Java");          names.add("C++");          names.add("Phython");          names.add("JavaScript");            List < String > first2Names = names.subList(0, 2);            System.out.println(names + " , " + first2Names);            names.set(1, "Ruby");            // check the output below.          System.out.println(names + " , " + first2Names);            // Now we add another string to          // get ConcurrentModificationException          names.add("SQL");            // This line throws an exception          System.out.println(names + " , " + first2Names);        }  } |

**Output:**



**Why we cannot create a generic array in Java?**

Java does not allow creation of array with generics as elements. In Java an array has to know the type information of its elements at runtime. This information is used at runtime to throw ArrayStoreException if data type of an element to be inserted does not match the type of Array. In case of Generics, the type information of a collection is erased at runtime by Type Erasure. Due to this array cannot use generics as elements

**Can you explain how HashMap works in Java?**

In Java, a HashMap works on the concept of hashing. A HashMap in Java stores both key and value objects, in a bucket. It is stored as an Entry object that implements Map.Entry interface. The key object used in a HashMap has to provide implementation for hashCode() and equals() methods. When put() method is used to store a key-value pair, the HashMap implementation calls hashCode() method on Key object to calculate a hash that is used to find a bucket where Entry object will be stored. When get() method is used to retrieve a value stored against a key object, we first calculate a hash of Key object. Then we use this hash to find the bucket in which that particular key is stored. Once Key object’s location is found, it may happen that more than one Key is stored in same location. So now we use equals() method to find the exact Key object. Once the exact Key object is found we use it to get Value object.

import java.util.\* ;

class HelloWorld {

public static int sumOfTwo(int[] a)

{

int sum = 0;

int count = 0 ;

for(int i=0 ; i <a.length ; i++ )

{

if(count != 2)

{

if(a[i]<0)

{

sum = sum + a[i];

count ++ ;

}

}

}

return sum ;

}

public static void main(String[] args) {

//int n1 = sc.nextInt();

int n1 = 10;

int n2 = 9 ;

int n3 = 6 ;

int n4 = 1;

int[] a = new int[4];

a[0] = n1 ;

a[1] = n2 ;

a[2] = n3 ;

a[3] = n4 ;

System.out.println("Sum of negative numbers : "+sumOfTwo(a));

}

**}**

**Can we mark a constructor as synchronized in Java?**

No. We cannot mark a constructor as synchronized. This will lead to compiler error. The reasoning behind this is that, in this case, only the constructing thread would have access to the object being constructed.

**What is the data type of a Lambda expression?**

A Lambda expression fulfills the purpose of passing code as data. The data type of a Lambda expression is a Functional interface. In most of the cases this is java.lang.Runnable interface.

# Volatile Keyword in Java

Volatile keyword is used to modify the value of a variable by different threads. It is also used to make classes thread safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

The volatile keyword does not cache the value of the variable and always read the variable from the main memory. The volatile keyword cannot be used with classes or methods. However, it is used with variables. It also guarantees visibility and ordering. It prevents the compiler from the reordering of code.

The contents of the particular device register could change at any time, so you need the volatile keyword to ensure that such accesses are not optimized away by the compiler.

### Example

1. **class** Test
2. {
3. **static** **int** var=5;
4. }

In the above example, assume that two threads are working on the same class. Both threads run on different processors where each thread has its local copy of var. If any thread modifies its value, the change will not reflect in the original one in the main memory. It leads to data inconsistency because the other thread is not aware of the modified value.

47.2M

802

History of Java

1. **class** Test
2. {
3. **static** **volatile** **int** var =5;
4. }

In the above example, static variables are class members that are shared among all objects. There is only one copy in the main memory. The value of a volatile variable will never be stored in the cache. All read and write will be done from and to the main memory.

## When to use it?

* You can use a volatile variable if you want to read and write long and double variable automatically.
* It can be used as an alternative way of achieving synchronization in Java.
* All reader threads will see the updated value of the volatile variable after completing the write operation. If you are not using the volatile keyword, different reader thread may see different values.
* It is used to inform the compiler that multiple threads will access a particular statement. It prevents the compiler from doing any reordering or any optimization.
* If you do not use volatile variable compiler can reorder the code, free to write in cache value of volatile variable instead of reading from the main memory.

## Important points

* You can use the volatile keyword with variables. Using volatile keyword with classes and methods is illegal.
* It guarantees that value of the volatile variable will always be read from the main memory, not from the local thread cache.
* If you declared variable as volatile, Read and Writes are atomic
* It reduces the risk of memory consistency error.
* Any write to volatile variable in Java establishes a happen before the relationship with successive reads of that same variable.
* The volatile variables are always visible to other threads.
* The volatile variable that is an object reference may be null.
* When a variable is not shared between multiple threads, you do not need to use the volatile keyword with that variable.

## Difference between synchronization and volatile keyword

Volatile keyword is not a substitute of synchronized keyword, but it can be used as an alternative in certain cases. There are the following differences are as follows:

|  |  |
| --- | --- |
| **Volatile Keyword** | **Synchronization Keyword** |
| Volatile keyword is a field modifier. | Synchronized keyword modifies code blocks and methods. |
| The thread cannot be blocked for waiting in case of volatile. | Threads can be blocked for waiting in case of synchronized. |
| It improves thread performance. | Synchronized methods degrade the thread performance. |
| It synchronizes the value of one variable at a time between thread memory and main memory. | It synchronizes the value of all variables between thread memory and main memory. |
| Volatile fields are not subject to compiler optimization. | Synchronize is subject to compiler optimization. |

### Example of Volatile Keyword

In the following example, we have defined a class which increases the counter value. The run () method in the VolatileThread.java gets the updated value and old value when the thread begins execution. In the main class, we define the array of thread.

**VolatileData.java**

1. **public** **class** VolatileData
2. {
3. **private** **volatile** **int** counter = 0;
4. **public** **int** getCounter()
5. {
6. **return** counter;
7. }
8. **public** **void** increaseCounter()
9. {
10. ++counter;      //increases the value of counter by 1
11. }
12. }

**VolatileThread.java**

1. VolatileThread.java
2. **public** **class** VolatileThread **extends** Thread
3. {
4. **private** **final** VolatileData data;
5. **public** VolatileThread(VolatileData data)
6. {
7. **this**.data = data;
8. }
9. @Override
10. **public** **void** run()
11. {
12. **int** oldValue = data.getCounter();
13. System.out.println("[Thread " + Thread.currentThread().getId() + "]: Old value = " + oldValue);
14. data.increaseCounter();
15. **int** newValue = data.getCounter();
16. System.out.println("[Thread " + Thread.currentThread().getId() + "]: New value = " + newValue);
17. }
18. }

**VolatileMain.java**

1. **public** **class** VolatileMain
2. {
4. **private** **final** **static** **int** noOfThreads = 2;
5. **public** **static** **void** main(String[] args) **throws** InterruptedException
6. {
7. VolatileData volatileData = **new** VolatileData();     //object of VolatileData class
8. Thread[] threads = **new** Thread[noOfThreads];     //creating Thread array
9. **for**(**int** i = 0; i < noOfThreads; ++i)
10. threads[i] = **new** VolatileThread(volatileData);
11. **for**(**int** i = 0; i < noOfThreads; ++i)
12. threads[i].start();                 //starts all reader threads
13. **for**(**int** i = 0; i < noOfThreads; ++i)
14. threads[i].join();                  //wait for all threads
15. }
16. }

**Output:**

[Thread 9]: Old value = 0

[Thread 9]: New value = 1

[Thread 10]: Old value = 1

[Thread 10]: New value = 2

|  |  |
| --- | --- |
| **PATH** | **CLASSPATH** |
| PATH is an environment variable. | CLASSPATH is also an environment variable. |
| It is used by the operating system to find the executable files (.exe). | It is used by Application ClassLoader to locate the .class file. |
| You are required to include the directory which contains .exe files. | You are required to include all the directories which contain .class and JAR files. |
| PATH environment variable once set, cannot be overridden. | The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command. |

## How to Set CLASSPATH in Windows Using Command Prompt

Type the following command in your Command Prompt and press enter.

1. set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\jre1.8\rt.jar;

In the above command, The set is an internal DOS command that allows the

* 1. **what does JDK contain?**

**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.

Now we need an environment to make a run of our program. Henceforth,**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*.

Now let us discuss **JVM**, which stands out for java virtual machines. It is as follows:

* A **specification** where the working of Java Virtual Machine is specified. But implementation provider is independent to choose the algorithm. Its implementation has been provided by Sun and other companies.
* An **implementation** is a computer program that meets the requirements of the JVM specification.
* **Runtime Instance** Whenever you write a java command on the command prompt to run the java class, an instance of JVM is created.

Before proceeding to the differences between JDK, JRE, and JVM, let us discuss them in brief first and interrelate them with the image below proposed.



Don’t get confused as we are going to discuss all of them one by one.

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

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

**2. JRE** (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** the java program(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system.

**3.**[**JVM** (**Java Virtual Machine)**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an [**i*nterpreter***](https://www.geeksforgeeks.org/compiler-vs-interpreter-2/)**.**

Now let us discuss the components of JRE in order to understand its importance of it and perceive how it actually works. For this let us discuss components.

The components of JRE are as follows:

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

After having an adequate understanding of the components, now let us discuss the working of JDK. In order to understand how JDK works, let us consider an illustration below as follows:

**Illustration:**

*Consider a java source file saved as ‘Example.java’. The file is compiled into a set of Byte Code that is stored in a “.class” file. Here it will be “Example.class“.*



***Note****: From above, media operation computing during the compile time can be interpreted.*

*The following actions occur at runtime as listed below:*

* *Class Loader*
* *Byte Code Verifier*
* *Interpreter*
  + *Execute the Byte Code*
  + *Make appropriate calls to the underlying hardware*

**Now let us discuss in brief how JVM works out. It is as follows:**

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

***It is mainly responsible for three activities.***

* Loading
* Linking
* Initialization

**Similarly, now let us discuss the working of JRE which is as follows:**

* JVM(Java Virtual Machine) acts as a run-time engine to run Java applications. JVM is the one that actually calls the **main** method present in a java code. JVM is a part of JRE(Java Runtime Environment).
* Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustments. This is all possible because of JVM.
* When we compile a *.java* file, *.class* files(contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

**what is binding? how many types of bindings are there?**

Binding is a mechanism creating link between method call and method actual implementation. As per the polymorphism concept in Java , object can have many different forms. Object forms can be resolved at compile time and run time. If linking between method call  and method implementation is resolved at compile time then we call it static binding or  If it is resolved at run time then it dynamic binding. Dynamic binding uses object to resolve binding but static binding use type of the class and fields.

| **Sr. No.** | **Key** | **Static Binding** | **Dynamic Binding** |
| --- | --- | --- | --- |
| 1 | Basic | It is resolved at compile time | It is resolved at run time |
| 2 | Resolve mechanism | static binding use type of the class and fields | Dynamic binding uses object to resolve binding |
| 3 | Example | Overloading is an example of static binding | Method overriding is the example of Dynamic binding |
| 4. | Type of Methods | private, final and static methods and variables uses static binding | Virtual methods use dynamic binding |

| **Static Binding** | **Dynamic Binding** |
| --- | --- |
| It takes place at compile time for which is referred to as early binding | It takes place at runtime so do it is referred to as late binding. |
| It uses overloading more precisely operator overloading method | It uses overriding methods. |
| It takes place using normal functions | It takes place using virtual functions |
| Real objects never use static binding | Real objects use dynamic binding. |

* 1. **What is ARM ?**

Java provides a feature to make the code more robust and to cut down the lines of code. This feature is known as Automatic Resource Management(ARM) using **try-with-resources** from Java 7 onwards. The try-with-resources statement is a try statement that declares one or more resources.

This statement ensures that each resource is closed at the end of the statement, which eases working with external resources that need to be disposed of or closed in case of errors or successful completion of a code block.

**Resource** is an object that must be closed after the program is finished using it. Any object that implements **java.lang.AutoCloseable**, which includes all objects which implement java.io.Closeable, can be used as a resource.

### An old method of resource cleanup – Using finally

In earlier versions of Java before JDK 1.7, the closing of resources was done using the finally block.

Ex :

static String readFirstLineFromFile(String path) throws IOException

{

try (BufferedReader br = new BufferedReader(new FileReader(path)))

{

return br.readLine();

}

}

**Automatic Resource Management in multiple resources**

Multiple resources can be used inside a try-with-resources block and have them all automatically closed. In this case, the resources will be closed in the reverse order in which they were created inside the brackets.

* Java

|  |
| --- |
| // Java program to illustrate  // Automatic Resource Management  // in Java with multiple resource  class Resource {      public static void main(String s[])      {          // note the order of opening the resources          try (Demo d = new Demo(); Demo1 d1 = new Demo1()) {              int x = 10 / 0;              d.show();              d1.show1();          }          catch (ArithmeticException e) {              System.out.println(e);          }      }  }  // custom resource 1  class Demo implements AutoCloseable {      void show() { System.out.println("inside show"); }      public void close()      {          System.out.println("close from demo");      }  }  // custom resource 2  class Demo1 implements AutoCloseable {      void show1() { System.out.println("inside show1"); }      public void close()      {          System.out.println("close from demo1");      }  } |

**Output:**

close from demo1

close from demo

***Note:****In the above example, Demo and Demo1 are the custom resources managed inside the try block. Such resources need to implement the AutoCloseable interface. When we open any such AutoCloseable resource in the special try-with-resource block, immediately after finishing the try block, JVM calls this.close() method on all resources initialized in the try block.*

**Important Points:**

1. The finally blocks were used to clean up the resources before Java 7.
2. After java 7, resource cleanup is done automatically.
3. ARM is done when you initialize resources in the try-with-resources block because of the interface AutoCloseable. Its close method is invoked by JVM as soon as the try block finishes.
4. Calling the close() method might lead to unexpected results.
5. A resource that we use in try-with-resource must be subtypes of AutoCloseable to avoid a compile-time error.
6. The resources which are used in multiple resource ARM must be closed in reverse order as given in the above example
   1. **when to use abstract class and when to use interface**

**When to use an interface**

If the functionality we are creating will be useful across a wide range of disparate objects, use an interface. Abstract classes should be used primarily for objects that are closely related, whereas interfaces are best suited for providing a common functionality to unrelated classes.

Interfaces are a good choice when we think that the API will not change for a while.

Interfaces are also good when we want to have something similar to multiple inheritances since we can implement multiple interfaces.

If we are designing small, concise bits of functionality, use interfaces. If we are designing large functional units, use an abstract class.

**When to use an abstract class**

An abstract class is a good choice if we are using the inheritance concept since it provides a common base class implementation to derived classes.

An abstract class is also good if we want to declare non-public members. In an interface, all methods must be public.

If we want to add new methods in the future, then an abstract class is a better choice. Because if we add new methods to an interface, then all of the classes that already implemented that interface will have to be changed to implement the new methods.

If we want to create multiple versions of our component, create an abstract class. Abstract classes provide a simple and easy way to version our components. By updating the base class, all inheriting classes are automatically updated with the change. Interfaces, on the other hand, cannot be changed once created. If a new version of an interface is required, we must create a whole new interface.

Abstract classes have the advantage of allowing better forward compatibility. Once clients use an interface, we cannot change it; if they use an abstract class, we can still add behavior without breaking the existing code.

If we want to provide common, implemented functionality among all implementations of our component, use an abstract class. Abstract classes allow us to partially implement our class, whereas interfaces contain no implementation for any members.

* 1. **what does jar file contain?**

A [JAR (Java Archive)](https://www.geeksforgeeks.org/jar-files-java/) is a package file format typically used to aggregate many Java class files and associated metadata and resources (text, images, etc.) into one file to distribute application software or libraries on the Java platform.   
In simple words, a JAR file is a file that contains a compressed version of .class files, audio files, image files, or directories. We can imagine a .jar file as a zipped file(.zip) that is created by using WinZip software. Even, WinZip software can be used to extract the contents of a .jar . So you can use them for tasks such as lossless data compression, archiving, decompression, and archive unpacking

**Sleep():** This Method is used to pause the execution of current thread for a specified time in Milliseconds. Here, Thread does not lose its ownership of the monitor and resume’s it’s execution

**Wait():** This method is defined in object class. It tells the calling thread (a.k.a Current Thread) to wait until another thread invoke’s the notify() or notifyAll() method for this object, The thread waits until it reobtains the ownership of the monitor and Resume’s Execution.

| Wait() | Sleep() |
| --- | --- |
| Wait() method belongs to Object class. | Sleep() method belongs to Thread class. |
| Wait() method releases lock during Synchronization. | Sleep() method does not release the lock on object during Synchronization. |
| Wait() should be called only from Synchronized context. | There is no need to call sleep() from Synchronized context. |
| Wait() is not a static method. | Sleep() is a static method. |
| Wait() Has Three Overloaded Methods:   * wait() * wait(long timeout) * wait(long timeout, int nanos) | Sleep() Has Two Overloaded Methods:   * sleep(long millis)millis: milliseconds * sleep(long millis,int nanos) nanos: Nanoseconds |
| public final void wait(long timeout) | public static void sleep(long millis) throws Interrupted\_Execption |



**what are the things developer needs to share with client?**

**explain "System.out.println"**

**what happens when u create an object.**

**what is constructor? types. what is special about default constructor?**

**what is "this" and what is its importance?**

**what is static block? Where have u seen an application of static block inside JDK?**

what is upcasting? what is the purpose of upcasting?

how many types of inheritance are alld in java? describe them

what is downcasting? what is the need of downcasting?

what is ClassCastException? what is its solution ?

what are the scenarios in which we have early binding in java?

program to implementation vs interface explain with example.

how does co-variant return type works in java ?

checked vs unchecked exceptions

handle vs declare rule

what is finally block ? what is its use? what r the scenarios where finally block does not get executed?

what is ARM

access modifiers in java

boxing and unboxing. ltd only to compiler level

explain "extends Thread"

explain "implements Runnable"

what is synchronization

what happens when u use synchronization

what is class lock

what is Thread Safe class

What is ReEntrantLock? what are the pros and cons of using it.

wait vs sleep

rules for serialization

how does put work in HashMap

how does get work in HashMap

if you want to store object of any user defined class as a key inside TreeMap or as an element inside TreeSet, what care u will take?

how does set work internally

what is Fail-Fast and Fail-safe iterator

difference between Hashtable,HashMap and ConcurrentHashMap

how generics are type-safe

? extends FourWheeler vs ? super Car

List<?> vs List<Object>

what is Class-Wins Rule in Java8

some times synchronized collections are not sufficient why

Iterator itr=mylist.iterator()

how does it work?

how do u iterate on the map