Java Arrays

Normally, an array is a collection of similar type of elements which has contiguous memory location.

**Java array** is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

### **Advantages**

* **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
* **Random access:** We can get any data located at an index position.

### **Disadvantages**

* **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

\*\* The **Scanner class** in Java, found in the java.util package, is commonly used for obtaining input of primitive types like int, double, and strings. While it’s straightforward for reading input in a Java program, there are some **disadvantages** associated with using it:

1. **Inefficiency in Competitive Programming**:
   * The Scanner class is **not very efficient** when time is a constraint, especially in competitive programming scenarios. If you need a faster input method, consider alternatives.
2. **Slower Performance with Large Data**:
   * When dealing with **large amounts of data**, the Scanner class can be slower compared to other input methods. If performance is critical, explore more efficient options.
3. **Exception Handling for Mismatched Input**:
   * The Scanner class can throw exceptions if the input does not match the expected type. For example, if you use nextInt() and the user enters a string, an exception occurs.
4. **Not Synchronized and Not Thread-Safe**:
   * The Scanner class is **not synchronized** and **not thread-safe**. If you’re working in a multithreaded environment, be cautious.
5. **Limited Buffer Memory**:
   * The Scanner class has a relatively small buffer memory (approximately 1KB). If you’re dealing with large input streams, this limitation may impact performance.

## \*\*\*Java Classes

A class in Java is a set of objects which shares common characteristics/ behavior and common properties/ attributes. It is a user-defined blueprint or prototype from which objects are created. For example, Student is a class while a particular student named Ravi is an object.

### Properties of Java Classes

1. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
2. Class does not occupy memory.
3. Class is a group of variables of different data types and a group of methods.
4. A Class in Java can contain:
   * Data member
   * Method
   * Constructor
   * Nested Class
   * Interface

### Class Declaration in Java

access\_modifier **class** <class\_name>

{

data member;

method;

constructor;

nested class;

interface;

}

### Components of Java Classes

 In general, class declarations can include these components, in order:

1. ***Modifiers****: A class can be public or has default access (Refer*[*this*](https://www.geeksforgeeks.org/access-specifiers-for-classes-or-interfaces-in-java/)*for details).*
2. ***Class keyword:****class keyword is used to create a class.*
3. ***Class name:****The name should begin with an initial letter (capitalized by convention).*
4. ***Superclass(if any):****The name of the class’s parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.*
5. ***Interfaces(if any):****A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.*
6. ***Body:****The class body is surrounded by braces, { }.*

## \*\* Java Objects

An object in Java is a basic unit of Object-Oriented Programming and represents real-life entities. Objects are the instances of a class that are created to use the attributes and methods of a class.  A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

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

Example of an object: dog

\*\* ***Note:****When we create an object which is a non primitive data type, it’s always allocated on the heap memory.*

## \*\* Anonymous Objects in Java

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

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

## \*\*\*\*What are Constructors in Java?

In Java, a Constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling the constructor, memory for the object is allocated in the memory. It is a special type of method that is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

\****Note:****It is not necessary to write a constructor for a class. It is because the java compiler creates a default constructor (constructor with no arguments) if your class doesn’t have any.*

### **How Java Constructors are Different From Java Methods?**

* Constructors must have the same name as the class within which it is defined it is not necessary for the method in Java.
* Constructors do not return any type while method(s) have the return type or **void** if does not return any value.
* Constructors are called only once at the time of Object creation while method(s) can be called any number of times.

## \*\***Need of Constructors in Java**

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in the computer’s memory), then can a box be there with no value defined for its dimensions? The answer is N**o**.   
So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

### **When Java Constructor is called?**

Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members**of the same class. Rules for writing constructors are as follows:

* The constructor(s) of a class must have the same name as the class name in which it resides.
* A constructor in Java can not be abstract, final, static, or Synchronized.
* Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

So by far, we have learned constructors are used to initialize the object’s state. Like [methods](https://www.geeksforgeeks.org/methods-in-java/), a constructor also contains a collection of statements(i.e. instructions) that are executed at the time of Object creation.

## Types of Constructors in Java

Now is the correct time to discuss the types of the constructor, so primarily there are three types of constructors in Java are mentioned below:

* Default Constructor
* Parameterized Constructor
* Copy Constructor

### **1. Default Constructor in Java**

A constructor that has no parameters is known as default the constructor. A default constructor is invisible. And if we write a constructor with no arguments, the compiler does not create a default constructor. It is taken out. It is being overloaded and called a parameterized constructor. The default constructor changed into the parameterized constructor. But Parameterized constructor can’t change the default constructor.

\*\****Note:****Default constructor provides the default values to the object like 0, null, etc. depending on the type.*

### **2. Parameterized Constructor in Java**

A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

\*\****Remember: Does constructor return any value?***

*There are no “return value” statements in the constructor, but the constructor returns the current class instance. We can write ‘return’ inside a constructor.*

### 3. Copy Constructor in Java

Unlike other constructors copy constructor is passed with another object which copies the data available from the passed object to the newly created object.

***Note:****In Java,there is no such inbuilt copy constructor available like in other programming languages such as C++, instead we can create our own copy constructor by passing the object of the same class to the other instance(object) of the class.*

\*\*advantages of constructor in java

* + - **Initializing Object State**:
      * Constructors allow us to set up the initial state of an object. Just like dressing a newborn in their first outfit, constructors initialize an object with default or specific values.
      * When you create a new instance of a class, the constructor ensures that the object starts with the correct initial properties.
    - **Ensuring Proper Object Creation**:
      * Constructors act as quality control for object creation.
      * They ensure that every new object is correctly initialized and ready for use.
      * By defining constructors, we can enforce specific rules or conditions during object instantiation.
    - **Dependency Information**:
      * Constructors provide a way to express dependencies.
      * When we define a constructor, we can specify what the class requires to function properly.
      * By examining the constructor, other developers can understand the necessary dependencies for using the class.
    - **Types of Constructors**:
      * **Parameterized Constructor**:
        + Accepts arguments (parameters) and dynamically initializes instance variables with specified values during object creation.

**Default Constructor**:

* Does not accept any parameters.
* Initializes instance variables with certain default values.
* Every Java class has an invisible default constructor, which is automatically created unless we define a parameterized constructor.

\*\*\* constructor in java disadvantages

1. **No Copy Constructor Support**:
   * Unlike some other programming languages, Java constructors **cannot be directly copied**. [This limitation makes it challenging to duplicate objects without manually copying each field1](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/).
   * Imagine trying to clone a complex object without a straightforward way to create a new instance with the same state.
2. **Limited Exception Handling**:
   * When a constructor **throws an exception**, it cannot be caught and handled during object creation.
   * [This lack of exception handling can lead to issues if the constructor fails to set up the object properly1](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/).
   * Proper error handling becomes crucial to avoid unexpected behavior.
3. **No Return Values**:
   * Constructors do not return any values to indicate success or failure.
   * [Other methods must check whether an object was set up correctly after invoking the constructor](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/)[1](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/).
   * This design choice simplifies the constructor’s purpose but requires additional checks elsewhere.
4. **Cannot Be Inherited**:
   * Unlike methods, constructors are **not part of a class’s hierarchy**.
   * Subclasses cannot directly use the superclass’s constructors.
   * [This limitation affects the inheritance model in Java](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/)[1](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/).
5. **Overloading Complexity**:
   * Having multiple constructors with different arguments can become **tricky to manage**.
   * [Especially when constructors have a similar number and type of parameters, overloading can lead to confusion](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/)[1](https://aspiringyouths.com/advantages-disadvantages/constructor-in-java/).
   * Balancing flexibility with readability is essential

# Inheritance in Java

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

## Why Do We Need Java Inheritance?

* **Code Reusability:**The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
* **Method Overriding:**[Method Overriding](https://www.geeksforgeeks.org/overriding-in-java/) is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.
* **Abstraction:**The concept of abstract where we do not have to provide all details is achieved through inheritance. [Abstraction](https://www.geeksforgeeks.org/abstraction-in-java-2/)only shows the functionality to the user.

### **Important Terminologies Used in Java Inheritance**

* **Class:**Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
* **Super Class/Parent Class:**The class whose features are inherited is known as a superclass(or a base class or a parent class).
* **Sub Class/Child Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
* **Reusability:**Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

## **How to Use Inheritance in Java?**

The **extends keyword**is used for inheritance in Java. Using the extends keyword indicates you are derived from an existing class. In other words, “extends” refers to increased functionality.

**Syntax :**

class DerivedClass extends BaseClass   
{   
 //methods and fields   
}

**import** **java.io.\***;

*// Base or Super Class*

**class** **Employee** {

int salary = 60000;

}

*// Inherited or Sub Class*

**class** **Engineer** **extends** Employee {

int benefits = 10000;

}

*// Driver Class*

**class** **Gfg** {

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

{

Engineer E1 = **new** Engineer();

System.out.println("Salary : " + E1.salary

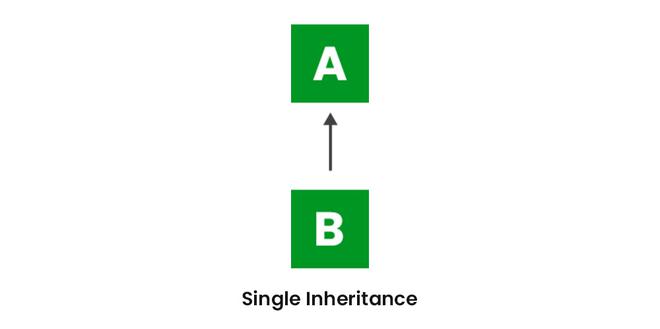
+ "\nBenefits : " + E1.benefits);

}

}

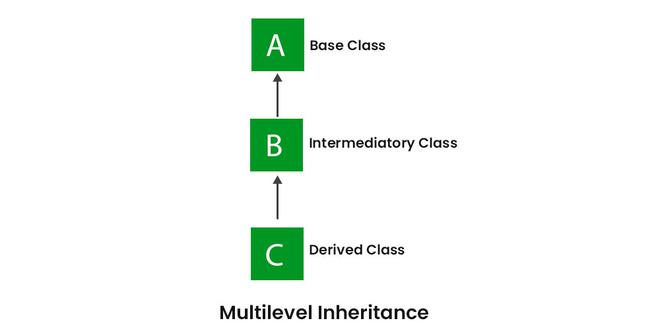
### **1. Single Inheritance**

In single inheritance, subclasses inherit the features of one superclass. In the image below, class A serves as a base class for the derived class B.



### **2. Multilevel Inheritance**

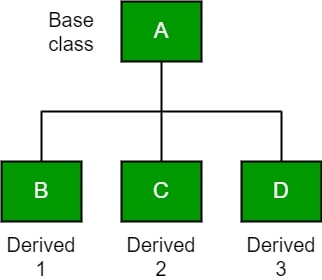
In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C. In Java, a class cannot directly access the[grandparent’s members](https://www.geeksforgeeks.org/g-fact-91/).



*Multilevel*

### **Hierarchical Inheritance**

In Hierarchical Inheritance, one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

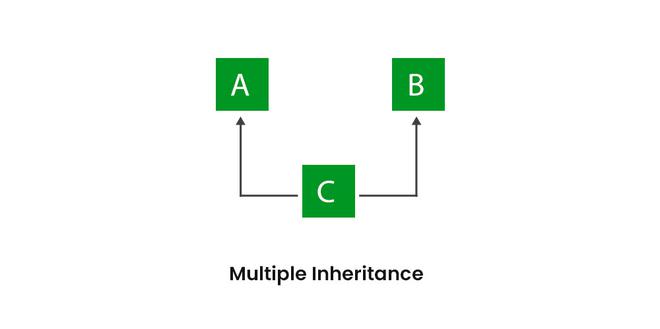


Java

*// Java program to illustrate the*

### 4. Multiple Inheritance (**Through Interfaces)**

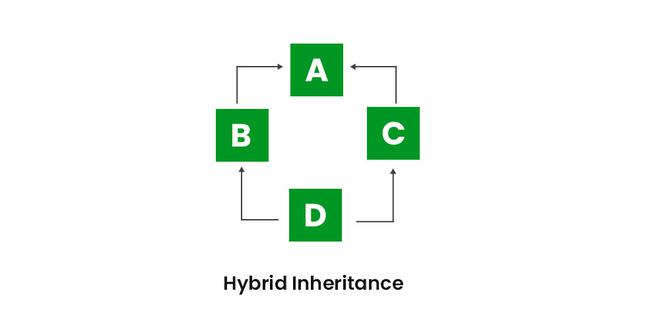
In [Multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/), one class can have more than one superclass and inherit features from all parent classes. Please note that Java does **not** support [multiple inheritances](https://www.geeksforgeeks.org/java-and-multiple-inheritance/) with classes. In Java, we can achieve multiple inheritances only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/). In the image below, Class C is derived from interfaces A and B.



*Multiple Inheritance*

### **5. Hybrid Inheritance**

It is a mix of two or more of the above types of inheritance. Since Java doesn’t support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through [Interfaces](https://www.geeksforgeeks.org/interfaces-in-java/) if we want to involve multiple inheritance to implement Hybrid inheritance.  
However, it is important to note that Hybrid inheritance does not necessarily require the use of Multiple Inheritance exclusively. It can be achieved through a combination of Multilevel Inheritance and Hierarchical Inheritance with classes, Hierarchical and Single Inheritance with classes. Therefore, it is indeed possible to implement Hybrid inheritance using classes alone, without relying on multiple inheritance type.



## **Java IS-A type of Relationship**

IS-A is a way of saying: This object is a type of that object. Let us see how the extends keyword is used to achieve inheritance.

## **What Can Be Done in a Subclass?**

In sub-classes we can inherit members as is, replace them, hide them, or supplement them with new members:

* The inherited fields can be used directly, just like any other fields.
* We can declare new fields in the subclass that are not in the superclass.
* The inherited methods can be used directly as they are.
* We can write a new instance method in the subclass that has the same signature as the one in the superclass, thus [overriding](https://www.geeksforgeeks.org/overriding-in-java/) it (as in the example above, toString() method is overridden).
* We can write a new static method in the subclass that has the same signature as the one in the superclass, thus [hiding](https://www.geeksforgeeks.org/g-fact-63/) it.
* We can declare new methods in the subclass that are not in the superclass.
* We can write a subclass constructor that invokes the constructor of the superclass, either implicitly or by using the keyword [super](https://www.geeksforgeeks.org/super-keyword/).

### Advantages Of Inheritance in Java:

1. Code Reusability: Inheritance allows for code reuse and reduces the amount of code that needs to be written. The subclass can reuse the properties and methods of the superclass, reducing duplication of code.
2. Abstraction: Inheritance allows for the creation of abstract classes that define a common interface for a group of related classes. This promotes abstraction and encapsulation, making the code easier to maintain and extend.
3. Class Hierarchy: Inheritance allows for the creation of a class hierarchy, which can be used to model real-world objects and their relationships.
4. Polymorphism: Inheritance allows for polymorphism, which is the ability of an object to take on multiple forms. Subclasses can override the methods of the superclass, which allows them to change their behavior in different ways.

### Disadvantages of Inheritance in Java:

1. Complexity: Inheritance can make the code more complex and harder to understand. This is especially true if the inheritance hierarchy is deep or if multiple inheritances is used.
2. Tight Coupling: Inheritance creates a tight coupling between the superclass and subclass, making it difficult to make changes to the superclass without affecting the subclass.

# Access Modifiers in Java

### in Java, Access modifiers help to restrict the scope of a class, constructor, variable, method, or data member. It provides security, accessibility, etc to the user depending upon the access modifier used with the element**1. 1) Default Access Modifier**

When no access modifier is specified for a class, method, or data member – It is said to be having the **default** access modifier by default. The data members, classes, or methods that are not declared using any access modifiers i.e. having default access modifiers are accessible **only within the same package**.

### **2. Private Access Modifier**

The private access modifier is specified using the keyword **private**. The methods or data members declared as private are accessible only **within the class** in which they are declared.

* Any other **class of**the **same package will not be able to access** these members.
* Top-level classes or interfaces can not be declared as private because
  + private means “only visible within the enclosing class”.
  + protected means “only visible within the enclosing class and any subclasses”

### **3. Protected Access Modifier**

The protected access modifier is specified using the keyword **protected**.

The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

## **4) Public Access modifier**

The public access modifier is specified using the keyword **public**.

* The public access modifier has the **widest scope** among all other access modifiers.
* Classes, methods, or data members that are declared as public are **accessible from everywhere** in the program. There is no restriction on the scope of public data members.

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

# System.exit() in Java

Last Updated : 10 Jul, 2016

The **java.lang.System.exit()** method exits current program by terminating running Java virtual machine. This method takes a status code. A non-zero value of status code is generally used to indicate abnormal termination. This is similar [exit in C/C++](https://www.geeksforgeeks.org/understanding-exit-abort-and-assert/).

Following is the declaration for **java.lang.System.exit()** method:

public static void exit(int status)

**exit(0)** : Generally used to indicate successful termination.  
**exit(1) or exit(-1) or any other non-zero value** – Generally indicates unsuccessful termination.

**Note :**This method does not return any value.

# \*\*Static Blocks in Java

In simpler language whenever we use a [static keyword](https://www.geeksforgeeks.org/static-keyword-java/) and associate it to a block then that block is referred to as a static block. Unlike C++, Java supports a special block, called a static block (also called static clause) that can be used for static initialization of a class. This code inside the static block is executed only once: the first time the class is loaded into memory.

**Calling of static block in java?**

Now comes the point of how to call this static block. So in order to call any static block, there is no specified way as static block executes automatically when the class is loaded in memory. Refer to the below illustration for understanding how static block is called.

***Remember:****Static blocks can also be executed before constructors.*

# *\*\** Super Keyword in Java

the**super keyword in Java** is a reference variable that is used to refer to parent class when we’re working with objects. You need to know the basics of [Inheritance](https://www.geeksforgeeks.org/inheritance-in-java/)and [Polymorphism](https://www.geeksforgeeks.org/polymorphism-in-java/) to understand the Java super keyword.

## Characteristics of Super Keyword in Java

* \***super is used to call a superclass constructor:** When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the constructor of the parent class.
* **super is used to call a superclass method:** A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class’s implementation of the method in addition to its own.
* **super is used to access a superclass field:**A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class’s version of a field.
* **super must be the first statement in a constructor:**When calling a superclass constructor, the super() statement must be the first statement in the constructor of the subclass.
* **super cannot be used in a static context:** The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.
* **super is not required to call a superclass method:**While it is possible to use the super keyword to call a method in the parent class, it is not required. If a method is not overridden in the subclass, then calling it without the super keyword will invoke the parent class’s implementation.

*Overall, the****super keyword****is a powerful tool for subclassing in Java, allowing subclasses to inherit and build upon the functionality of their parent classes.*

## \*Advantages of Using Java Super Keyword

The **super keyword in Java** provides many advantages in object-oriented programming are as follows:

* **Enables reuse of code**: Using the super keyword allows subclasses to inherit functionality from their parent classes, which promotes the reuse of code and reduces duplication.
* **Supports polymorphism**: Because subclasses can override methods and access fields from their parent classes using super, polymorphism is possible. This allows for more flexible and extensible code.
* **Provides access to parent class behaviour**: Subclasses can access and use methods and fields defined in their parent classes through the super keyword, which allows them to take advantage of existing behaviour without having to reimplement it.
* **Allows for customization of behaviour:** By overriding methods and using super to call the parent implementation, subclasses can customize and extend the behaviour of their parent classes.
* **Facilitates abstraction and encapsulation:** The use of super promotes encapsulation and abstraction by allowing subclasses to focus on their behaviour while relying on the parent class to handle lower-level details.
* **Note :** constructor cannot inherit from parent class to child class
* **Relationship in java have two types :** 1) is – a relationship 2) has – a relationship

# this’ reference in Java

In Java, ‘this’ is a reference variable that refers to the current object, or can be said “this” in Java is a keyword that refers to the current object instance. It can be used to call current class methods and fields, to pass an instance of the current class as a parameter, and to differentiate between the local and instance variables. Using “this” reference can improve code readability and reduce naming conflicts.

## \*\* Methods to use ‘this’ in Java

Following are the ways to use the ‘this’ keyword in Java mentioned below:

* Using the ‘this’ keyword to refer to current class instance variables.
* Using this() to invoke the current class constructor
* Using ‘this’ keyword to return the current class instance
* Using ‘this’ keyword as the method parameter
* Using ‘this’ keyword to invoke the current class method
* Using ‘this’ keyword as an argument in the constructor call

### \*\* Advantages of using ‘this’ reference

There are certain advantages of using ‘this’ reference in Java as mentioned below:

1. It helps to distinguish between instance variables and local variables with the same name.
2. It can be used to pass the current object as an argument to another method.
3. It can be used to return the current object from a method.
4. It can be used to invoke a constructor from another overloaded constructor in the same class.

### \*\* Disadvantages of using ‘this’ reference

Although ‘this’ reference comes with many advantages there are certain disadvantages of also:

1. Overuse of this can make the code harder to read and understand.
2. Using this unnecessarily can add unnecessary overhead to the program.
3. Using this in a static context results in a compile-time error.
4. Overall, this keyword is a useful tool for working with objects in Java, but it should be used judiciously and only when necessary.

## \*\***What is a method in Java?**

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code. We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.

### **User-defined Method**

The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

### **Predefined Method**

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

# Method Overloading in Java

* If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**
* If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).

## **Advantage of method overloading**

Method overloading increases the readability of the program.

### **Different ways to overload the method**

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

### **Can we overload java main() method**

* Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only

## **Method Overloading and Type Promotion**

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

* 

# Method Overriding in Java

if subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

### **Usage of Java Method Overriding**

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### **Rules for Java Method Overriding**

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance).

### **Can we override static method?**

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

### **Why can we not override static method?**

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

### **Can we override java main method?**

No, because the main is a static method.

# Aggregation in Java

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

### **When use Aggregation?**

* Code reuse is also best achieved by aggregation when there is no is-a relationship.
* Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

### **Understanding meaningful example of Aggregation**

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

#### **Address.java**

1. **public** **class** Address {
2. String city,state,country;
4. **public** Address(String city, String state, String country) {
5. **this**.city = city;
6. **this**.state = state;
7. **this**.country = country;
8. }
10. }
11. **public** **class** Emp {
12. **int** id;
13. String name;
14. Address address;
16. **public** Emp(**int** id, String name,Address address) {
17. **this**.id = id;
18. **this**.name = name;
19. **this**.address=address;
20. }
22. **void** display(){
23. System.out.println(id+" "+name);
24. System.out.println(address.city+" "+address.state+" "+address.country);
25. }
27. **public** **static** **void** main(String[] args) {
28. Address address1=**new** Address("gzb","UP","india");
29. Address address2=**new** Address("gno","UP","india");
31. Emp e=**new** Emp(111,"varun",address1);
32. Emp e2=**new** Emp(112,"arun",address2);
34. e.display();
35. e2.display();
37. }
38. }

# Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type.

## **Advantages of Covariant Return Type**

Following are the advantages of the covariant return type.

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

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

3) Covariant return type helps in preventing the run-time ClassCastExceptions on returns.

# Instance initializer block

|  |
| --- |
| **Instance Initializer block** is used to initialize the instance data member. It run each time when object of the class is created. |
| The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block. |

## **Why use instance initializer block?**

|  |
| --- |
| Suppose I have to perform some operations while assigning value to instance data member e.g. a for loop to fill a complex array or error handling etc. |

### **Example of instance initializer block**

|  |
| --- |
| Let's see the simple example of instance initializer block that performs initialization. |

1. **class** Bike7{
2. **int** speed;
4. Bike7(){System.out.println("speed is "+speed);}
6. {speed=100;}
8. **public** **static** **void** main(String args[]){
9. Bike7 b1=**new** Bike7();
10. Bike7 b2=**new** Bike7();
11. }
12. }

#### **Note: The java compiler copies the code of instance initializer block in every constructor.**

## **Rules for instance initializer block :**

|  |
| --- |
| There are mainly three rules for the instance initializer block. They are as follows: |

1. The instance initializer block is created when instance of the class is created.
2. The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).
3. The instance initializer block comes in the order in which they appear.

# Final Keyword In Java

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

## **1) Java final variable**

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

## **2) Java final method**

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

## **3) Java final class**

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

### **) Is final method inherited?**

Ans) Yes, final method is inherited but you cannot override it

### **Que) Can we initialize blank final variable?**

Yes, but only in constructor.

### **static blank final variable**

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

### **Q) What is final parameter?**

If you declare any parameter as final, you cannot change the value of it.

### **Q) Can we declare a constructor final?**

No, because constructor is never inherited.

# Polymorphism in Java

**Polymorphism in Java** is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.

* There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism. We can perform polymorphism in java by method overloading and method overriding.
* \*\* **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

# Static Binding and Dynamic Binding

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

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

### **static binding**

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

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

### **Dynamic binding**

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

* Not -- > static keyword use with instance variable , cannot use with local variable

# Abstraction in Java

**Abstraction in Java** is the process in which we only show essential details/functionality to the user. The non-essential implementation details are not displayed to the user.

#### **Simple Example to understand Abstraction:**

***Television remote control****is an excellent****example of abstraction****. It simplifies the interaction with a TV by hiding the complexity behind simple buttons and symbols, making it easy without needing to understand the technical details of how the TV functions.*

## \*\* What is Abstraction in Java?

In Java, abstraction is achieved by[**interfaces**](https://www.geeksforgeeks.org/interfaces-in-java/)and [**abstract classes**](https://www.geeksforgeeks.org/abstract-classes-in-java/). We can achieve 100% abstraction using interfaces.

* Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details. The properties and behaviours of an object differentiate it from other objects of similar type and also help in classifying/grouping the objects.

## **Java Abstract classes and Java Abstract methods**

1. An abstract class is a class that is declared with an [abstract keyword.](https://www.geeksforgeeks.org/abstract-keyword-in-java/)
2. An abstract method is a method that is declared without implementation.
3. An abstract class may or may not have all abstract methods. Some of them can be concrete methods
4. A method-defined abstract must always be redefined in the subclass, thus making [overriding](https://www.geeksforgeeks.org/overriding-in-java/) compulsory or making the subclass itself abstract.
5. Any class that contains one or more abstract methods must also be declared with an abstract keyword.
6. There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the [new operator](https://www.geeksforgeeks.org/new-operator-java/).
7. An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

## Interface

Interfaces are another method of implementing abstraction in Java. The key difference is that, by using interfaces, we can achieve 100% abstraction in Java classes. In Java or any other language, interfaces include both methods and variables but lack a method body. Apart from abstraction, interfaces can also be used to implement interfaces in Java.

## **Advantages of Abstraction**

Here are some advantages of abstraction:

1. It reduces the complexity of viewing things.
2. Avoids code duplication and increases reusability.
3. Helps to increase the security of an application or program as only essential details are provided to the user.
4. It improves the maintainability of the application.
5. It improves the modularity of the application.
6. The enhancement will become very easy because without affecting end-users we can able to perform any type of changes in our internal system.
7. Improves code reusability and maintainability.
8. Hides implementation details and exposes only relevant information.
9. Provides a clear and simple interface to the user.
10. Increases security by preventing access to internal class details.
11. Supports modularity, as complex systems can be divided into smaller and more manageable parts.
12. Abstraction provides a way to hide the complexity of implementation details from the user, making it easier to understand and use.
13. Abstraction allows for flexibility in the implementation of a program, as changes to the underlying implementation details can be made without affecting the user-facing interface.
14. Abstraction enables modularity and separation of concerns, making code more maintainable and easier to debug.

## Disadvantages of Abstraction in Java

Here are the main disadvantages of abstraction in Java:

1. Abstraction can make it more difficult to understand how the system works.
2. It can lead to increased complexity, especially if not used properly.
3. This may limit the flexibility of the implementation.
4. Abstraction can add unnecessary complexity to code if not used appropriately, leading to increased development time and effort.
5. Abstraction can make it harder to debug and understand code, particularly for those unfamiliar with the abstraction layers and implementation details.
6. Overuse of abstraction can result in decreased performance due to the additional layers of code and indirection.

# \* Encapsulation in Java

* Encapsulation in Java is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that operate on that data within a single unit, which is called a class in Java. Java Encapsulation is a way of hiding the implementation details of a class from outside access and only exposing a public interface that can be used to interact with the class.

In Java, encapsulation is achieved by declaring the instance variables of a class as private, which means they can only be accessed within the class. To allow outside access to the instance variables, public methods called getters and setters are defined, which are used to retrieve and modify the values of the instance variables, respectively. By using getters and setters, the class can enforce its own data validation rules and ensure that its internal state remains consistent.

* **Encapsulation** is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates. Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.
* Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of its own class in which it is declared.
* As in encapsulation, the data in a class is hidden from other classes using the data hiding concept which is achieved by making the members or methods of a class private, and the class is exposed to the end-user or the world without providing any details behind implementation using the abstraction concept, so it is also known as a **combination of data-hiding and abstraction**.
* Encapsulation can be achieved by Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.
* It is more defined with the setter and getter method.

## **Advantages of Encapsulation**

* **Data Hiding:**it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
* **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirements. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
* **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
* **Testing code is easy:** Encapsulated code is easy to test for unit testing.
* **Freedom to programmer in implementing the details of the system:**This is one of the major advantage of encapsulation that it gives the programmer freedom in implementing the details of a system. The only constraint on the programmer is to maintain the abstract interface that outsiders see.

### Disadvantages of Encapsulation in Java

* Can lead to increased complexity, especially if not used properly.
* Can make it more difficult to understand how the system works.
* May limit the flexibility of the implementation.

# Why Java is not a purely Object-Oriented Language?

* Pure Object Oriented Language or Complete Object Oriented Language are Fully Object Oriented Language that supports or have features that treats everything inside the program as objects. It doesn’t support primitive datatype(like int, char, float, bool, etc.). There are seven qualities to be satisfied for a programming language to be pure object-oriented. They are:

1. Encapsulation/Data Hiding
2. Inheritance
3. Polymorphism
4. Abstraction
5. All predefined types are objects
6. All user defined types are objects
7. All operations performed on objects must be only through methods exposed at the objects.

* **Why Java is not a Pure Object Oriented Language?**

Java supports properties 1, 2, 3, 4 and 6 but fails to support properties 5 and 7 given above. Java language is not a Pure Object Oriented Language as it contains these properties:

* **Primitive Data Type ex. int, long, bool, float, char, etc as Objects:** Smalltalk is a “pure” object-oriented programming language unlike Java and C++ as there is no difference between values that are objects and values that are primitive types. In Smalltalk, primitive values such as integers, booleans, and characters are also objects. In Java, we have predefined types as non-objects (primitive types).

int a = 5;   
System.out.print(a);

* **The static keyword:**When we declare a class as static, then it can be used without the use of an object in Java. If we are using static function or static variable then we can’t call that function or variable by using dot(.) or class object defying object-oriented feature.
* **Wrapper Class:** Wrapper class provides the mechanism to convert primitive into object and object into primitive. In Java, you can use Integer, Float, etc. instead of int, float etc. We can communicate with objects without calling their methods. ex. using arithmetic operators.

String s1 = "ABC" + "A" ;

* Even using Wrapper classes does not make Java a pure OOP language, as internally it will use the operations like Unboxing and Autoboxing. So if you create Integer instead of int and do any mathematical operation on it, under the hoods Java is going to use primitive type int only.

# Exception Handling in Java

The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

## **What is Exception in Java?**

**Dictionary Meaning:** Exception is an abnormal condition.

In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

## **What is Exception Handling?**

Exception Handling is a mechanism to handle runtime errors such as ClassNotFoundException, IOException, SQLException, RemoteException, etc.

### **Advantage of Exception Handling**

The core advantage of exception handling is **to maintain the normal flow of the application**. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions.

## **Hierarchy of Java Exception classes**

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:



### **Types of Java Exceptions**

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception
2. Unchecked Exception
3. Error

## **\* Difference between Checked and Unchecked Exceptions**

### **1) Checked Exception**

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

### **2) Unchecked Exception**

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

### **3) Error**

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

## **Java Exception Keywords**

Java provides five keywords that are used to handle the exception. The following table describes each.

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| try | The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally. |
| catch | The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later. |
| finally | The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not. |
| throw | The "throw" keyword is used to throw an exception. |
| throws | The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature. |

# Java try-catch block

## **Java try block**

Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

### **Syntax of Java try-catch**

1. **try**{
2. //code that may throw an exception
3. }**catch**(Exception\_class\_Name ref){}

### **Syntax of try-finally block**

1. **try**{
2. //code that may throw an exception
3. }**finally**{}

## **Java catch block**

# Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception ( i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

## **Internal Working of Java try-catch block**



* The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:
* Prints out exception description.
* Prints the stack trace (Hierarchy of methods where the exception occurred).
* Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

# Java Catch Multiple Exceptions

## **Java Multi-catch block**

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

## **Points to remember**

* At a time only one exception occurs and at a time only one catch block is executed.
* All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

# Java Nested try block

In Java, using a try block inside another try block is permitted. It is called as nested try block. Every statement that we enter a statement in try block, context of that exception is pushed onto the stack.

For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

### **Why use nested try block**

Sometimes a situation may arise where a part of a block may cause one error and the entire block itself may cause another error. In such cases, exception handlers have to be nested.

### **Syntax:**

1. /main try block
2. **try**
3. {
4. statement 1;
5. statement 2;
6. //try catch block within another try block
7. **try**
8. {
9. statement 3;
10. statement 4;
11. //try catch block within nested try block
12. **try**
13. {
14. statement 5;
15. statement 6;
16. }
17. **catch**(Exception e2)
18. {
19. //exception message
20. }
22. }
23. **catch**(Exception e1)
24. {
25. //exception message
26. }
27. }
28. //catch block of parent (outer) try block
29. **catch**(Exception e3)
30. {
31. //exception message
32. }
33. ....

# Java finally block

**Java finally block** is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not. Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

## **Why use Java finally block?**

* finally block in Java can be used to put "**cleanup**" code such as closing a file, closing connection, etc.
* The important statements to be printed can be placed in the finally block.

# Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

## **Java throw keyword**

The Java throw keyword is used to throw an exception explicitly.

We specify the **exception** object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section.

We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. **throw** **new** exception\_class("error message");

Let's see the example of throw IOException.

1. **throw** **new** IOException("sorry device error");

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

# Java Exception Propagation

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

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

# Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

### **Syntax of Java throws**

1. return\_type method\_name() **throws** exception\_class\_name{
2. //method code
3. }

### **Which exception should be declared?**

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

* **unchecked exception:** under our control so we can correct our code.
* **error:** beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.

### **Advantage of Java throws keyword**

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

It provides information to the caller of the method about the exception.

# Difference between throw and throws in Java

The throw and throws is the concept of exception handling where the throw keyword throw the exception explicitly from a method or a block of code whereas the throws keyword is used in signature of the method.

There are many differences between [throw](https://www.javatpoint.com/throw-keyword) and [throws](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws) keywords. A list of differences between throw and throws are given below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. no.** | **Basis of Differences** | **throw** | **throws** |
| 1. | Definition | Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code. | Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code. |
| 2. | Usage | Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only. | Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only. |
| 3. | Syntax | The throw keyword is followed by an instance of Exception to be thrown. | The throws keyword is followed by class names of Exceptions to be thrown. |
| 4. | Declaration | throw is used within the method. | throws is used with the method signature. |
| 5. | Internal implementation | We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions. | We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException. |

* Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the [**final**](https://www.javatpoint.com/final-keyword) is an access modifier, [**finally**](https://www.javatpoint.com/finally-block-in-exception-handling) is the block in Exception Handling and [**finalize**](https://www.javatpoint.com/java-object-finalize-method) is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sr. no.** | **Key** | **final** | **finally** | **finalize** |
| 1. | Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |
| 2. | Applicable to | Final keyword is used with the classes, methods and variables. | Finally block is always related to the try and catch block in exception handling. | finalize() method is used with the objects. |
| 3. | Functionality | (1) Once declared, final variable becomes constant and cannot be modified. (2) final method cannot be overridden by sub class. (3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not. (2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| 4. | Execution | Final method is executed only when we call it. | Finally block is executed as soon as the try-catch block is executed.  It's execution is not dependant on the exception. | finalize method is executed just before the object is destroyed. |

# Exception Handling with Method Overriding in Java

There are many rules if we talk about method overriding with exception handling.

* **If the superclass method does not declare an exception**
  + If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but it can declare unchecked exception.
* **If the superclass method declares an exception**
  + If the superclass method declares an exception, subclass overridden method can declare same, subclass exception or no exception but cannot declare parent exception.

### **If the superclass method does not declare an exception**

#### **Rule 1: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception.**

#### **Rule 2: If the superclass method does not declare an exception, subclass overridden method cannot declare the checked exception but can declare unchecked exception.**

# java Custom Exception

In Java, we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Consider the example 1 in which InvalidAgeException class extends the Exception class.

Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.

## **Why use custom exceptions?**

Java exceptions cover almost all the general type of exceptions that may occur in the programming. However, we sometimes need to create custom exceptions.

* To catch and provide specific treatment to a subset of existing Java exceptions.
* Business logic exceptions: These are the exceptions related to business logic and workflow. It is useful for the application users or the developers to understand the exact problem.

#### **Note: We need to write the constructor that takes the String as the error message and it is called parent class constructor.**

# Java Inner Classes (Nested Classes)

**Java inner class** or nested class is a class that is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place to be more readable and maintainable.

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

### **Advantage of Java inner classes**

There are three advantages of inner classes in Java. They are as follows:

1. Nested classes represent a particular type of relationship that is **it can access all the members (data members and methods) of the outer class,** including private.
2. Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
3. **Code Optimization**: It requires less code to write.

## **Need of Java Inner class**

Sometimes users need to program a class in such a way so that no other class can access it. Therefore, it would be better if you include it within other classes.

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

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

# Java BufferedReader Class

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

## **Java BufferedReader class declaration**

Let's see the declaration for Java.io.BufferedReader class:

1. **public** **class** BufferedReader **extends** Reader

## **Reading data from console by InputStreamReader and BufferedReader**

In this example, we are connecting the BufferedReader stream with the [InputStreamReader](https://www.javatpoint.com/Input-from-keyboard-by-InputStreamReader) stream for reading the line by line data from the keyboard.

1. **package** com.javatpoint;
2. **import** java.io.\*;
3. **public** **class** BufferedReaderExample{
4. **public** **static** **void** main(String args[])**throws** Exception{
5. InputStreamReader r=**new** InputStreamReader(System.in);
6. BufferedReader br=**new** BufferedReader(r);
7. System.out.println("Enter your name");
8. String name=br.readLine();
9. System.out.println("Welcome "+name);
10. }
11. }

### **What is String in Java?**

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

### **How to create a string object?**

There are two ways to create String object:

1. By string literal
2. By new keyword

### **1) String Literal**

Java String literal is created by using double quotes

1. String s="welcome";

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

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



In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

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

### **Why Java uses the concept of String literal?**

To make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

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

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

In such case, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

### **java String class methods**

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

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

# Immutable String in Java

A String is an unavoidable type of variable while writing any application program. String references are used to store various attributes like username, password, etc. In Java, **String objects are immutable**. Immutable simply means unmodifiable or unchangeable.

### **Why String objects are immutable in Java?**

As Java uses the concept of String literal. Suppose there are 5 reference variables, all refer to one object "Sachin". If one reference variable changes the value of the object, it will be affected by all the reference variables. That is why String objects are immutable in Java.

Following are some features of String which makes String objects immutable.

**1. ClassLoader:**

A ClassLoader in Java uses a String object as an argument. Consider, if the String object is modifiable, the value might be changed and the class that is supposed to be loaded might be different.

To avoid this kind of misinterpretation, String is immutable.

**2. Thread Safe:**

As the String object is immutable we don't have to take care of the synchronization that is required while sharing an object across multiple threads.

**3. Security:**

As we have seen in class loading, immutable String objects avoid further errors by loading the correct class. This leads to making the application program more secure. Consider an example of banking software. The username and password cannot be modified by any intruder because String objects are immutable. This can make the application program more secure.

**4. Heap Space:**

The immutability of String helps to minimize the usage in the heap memory. When we try to declare a new String object, the JVM checks whether the value already exists in the String pool or not. If it exists, the same value is assigned to the new object. This feature allows Java to use the heap space efficiently.

### **Why String class is Final in Java?**

The reason behind the String class being final is because no one can override the methods of the String class. So that it can provide the same features to the new String objects as well as to the old ones

# Java StringBuilder Class

Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

## **Important Constructors of StringBuilder class**

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

## **Important methods of StringBuilder class**

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

# Java StringBuffer Class

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

#### **Note: Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.**

### **Important Constructors of StringBuffer Class**

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

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

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

# Thread Concept in Java

Before introducing the **thread concept**, we were unable to run more than one task in parallel. It was a drawback, and to remove that drawback, **Thread Concept** was introduced.

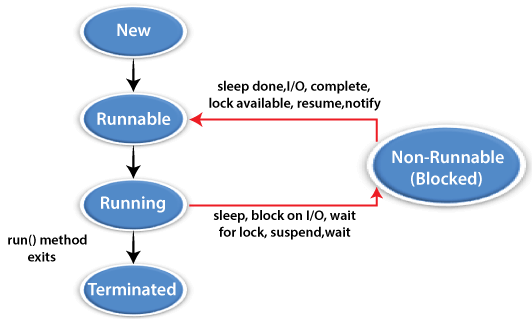
A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.

In order to perform complicated tasks in the background, we used the **Thread concept in Java**. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.

* Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as [**Multithreading**](https://www.javatpoint.com/multithreading-in-java).
* In a simple way, a Thread is a:
* Feature through which we can perform multiple activities within a single process.
* Lightweight process.
* Series of executed statements.
* Nested sequence of method calls.

## **Thread Model**

Just like a process, a thread exists in several states. These states are as follows:



**1) New (Ready to run)**

A thread is in **New** when it gets CPU time.

**2) Running**

A thread is in **a Running** state when it is under execution.

**3) Suspended**

A thread is in the **Suspended** state when it is temporarily inactive or under execution.

**4) Blocked**

A thread is in the **Blocked** state when it is waiting for resources.

**5) Terminated**

A thread comes in this state when at any given time, it halts its execution immediately.

## **Creating Thread**

A thread is created either by "creating or implementing" the **Runnable Interface** or by extending the **Thread class**. These are the only two ways through which we can create a thread.

### **Thread Class**

A **Thread class** has several methods and constructors which allow us to perform various operations on a thread. The Thread class extends the **Object** class. The **Object** class implements the **Runnable** interface. The thread class has the following constructors that are used to perform various operations.

* **Thread()**
* **Thread(Runnable, String name)**
* **Thread(Runnable target)**
* **Thread(ThreadGroup group, Runnable target, String name)**
* **Thread(ThreadGroup group, Runnable target)**
* **Thread(ThreadGroup group, String name)**
* **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**

### **Runnable Interface(run() method)**

The Runnable interface is required to be implemented by that class whose instances are intended to be executed by a thread. The runnable interface gives us the **run()** method to perform an action for the thread.

### **start() method**

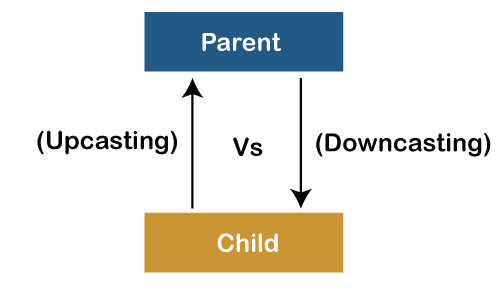
The method is used for starting a thread that we have newly created. It starts a new thread with a new callstack. After executing the **start()** method, the thread changes the state from New to Runnable. It executes the **run() method** when the thread gets the correct time to execute it.

# Upcasting and Downcasting in Java

A process of converting one data type to another is known as **Typecasting** and **Upcasting** and **Downcasting** is the type of object typecasting. In Java, the object can also be typecasted like the datatypes. **Parent** and **Child** objects are two types of objects. So, there are two types of typecasting possible for an object, i.e., **Parent to Child** and **Child to Parent** or can say **Upcasting** and **Downcasting**.

In [Java](https://www.javatpoint.com/java-tutorial), the object can also be typecasted like the datatypes. **Parent** and **Child objects** are two types of objects. So, there are two types of typecasting possible for an object, i.e., **Parent to Child** and **Child to Parent** or can say **Upcasting** and **Downcasting**.

**Typecasting** is used to ensure whether variables are correctly processed by a function or not. In **Upcasting** and **Downcasting**, we typecast **a child object to a parent object** and **a parent object to a child object** simultaneously. We can perform Upcasting implicitly or explicitly, but downcasting cannot be implicitly possible.



Let's dive into deep of both these type of object casting:

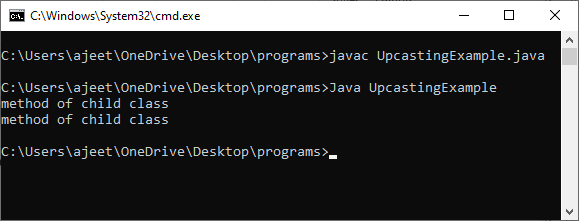
### **1) Upcasting**

**Upcasting** is a type of object typecasting in which a **child object** is typecasted to a **parent class object**. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. **Upcasting** is also known as **Generalization** and **Widening**.

**UpcastingExample.java**

1. **class**  Parent{
2. **void** PrintData() {
3. System.out.println("method of parent class");
4. }
5. }
7. **class** Child **extends** Parent {
8. **void** PrintData() {
9. System.out.println("method of child class");
10. }
11. }
12. **class** UpcastingExample{
13. **public** **static** **void** main(String args[]) {
15. Parent obj1 = (Parent) **new** Child();
16. Parent obj2 = (Parent) **new** Child();
17. obj1.PrintData();
18. obj2.PrintData();
19. }
20. }

**Output:**



### **2) Downcasting**

**Upcasting** is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the **"ClassCastException"**. Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

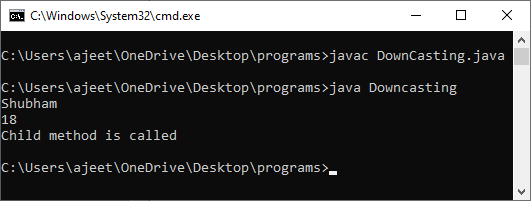
Below is an example of downcasting in which both the valid and the invalid scenarios are explained:

**DowncastingExample.java**

ADVERTISEMENT

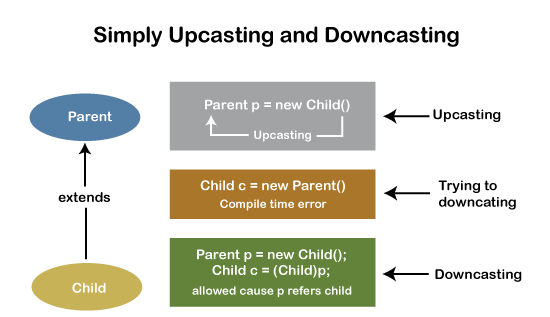
1. //Parent class
2. **class** Parent {
3. String name;
5. // A method which prints the data of the parent class
6. **void** showMessage()
7. {
8. System.out.println("Parent method is called");
9. }
10. }
12. // Child class
13. **class** Child **extends** Parent {
14. **int** age;
16. // Performing overriding
17. @Override
18. **void** showMessage()
19. {
20. System.out.println("Child method is called");
21. }
22. }
24. **public** **class** Downcasting{
26. **public** **static** **void** main(String[] args)
27. {
28. Parent p = **new** Child();
29. p.name = "Shubham";
31. // Performing Downcasting Implicitly
32. //Child c = new Parent(); // it gives compile-time error
34. // Performing Downcasting Explicitly
35. Child c = (Child)p;
37. c.age = 18;
38. System.out.println(c.name);
39. System.out.println(c.age);
40. c.showMessage();
41. }
42. }

**Output:**



### **Why we need Upcasting and Downcasting?**

In Java, we rarely use **Upcasting**. We use it when we need to develop a code that deals with only the parent class. **Downcasting** is used when we need to develop a code that accesses behaviors of the child class.



### **Difference between Upcasting and Downcasting**

These are the following differences between Upcasting and Downcasting:

|  |  |  |
| --- | --- | --- |
| **S.No** | **Upcasting** | **Downcasting** |
| 1. | A child object is typecasted to a parent object. | The reference of the parent class object is passed to the child class. |
| 2. | We can perform Upcasting implicitly or explicitly. | Implicitly Downcasting is not possible. |
| 3. | In the child class, we can access the methods and variables of the parent class. | The methods and variables of both the classes(parent and child) can be accessed. |
| 4. | We can access some specified methods of the child class. | All the methods and variables of both classes can be accessed by performing downcasting. |
| 5. | Parent p = new Parent() | Parent p = new Child() Child c = (Child)p; |

# Multithreading in Java

**Multithreading in**[**Java**](https://www.javatpoint.com/java-tutorial) is a process of executing multiple threads simultaneously.

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

However, we use multithreading than multiprocessing because threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

Java Multithreading is mostly used in games, animation, etc.

### **Advantages of Java Multithreading**

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

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

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

## **Multitasking**

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

* Process-based Multitasking (Multiprocessing)
* Thread-based Multitasking (Multithreading)

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

* Each process has an address in memory. In other words, each process allocates a separate memory area.
* A process is heavyweight.
* Cost of communication between the process is high.
* Switching from one process to another requires some time for saving and loading [registers](https://www.javatpoint.com/register-memory), memory maps, updating lists, etc.

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

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

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

## **What is Thread in java**

A thread is a lightweight subprocess, the smallest unit of processing. It is a separate path of execution.

Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

#### **Note: At a time one thread is executed only.**

## **Java Thread class**

Java provides **Thread class** to achieve thread programming. Thread class provides [constructors](https://www.javatpoint.com/java-constructor) and methods to create and perform operations on a thread. Thread class extends [Object class](https://www.javatpoint.com/object-class) and implements Runnable interface.

## **Java Thread Methods**

|  |  |  |  |
| --- | --- | --- | --- |
| **S.N.** | **Modifier and Type** | **Method** | **Description** |
| 1) | void | [start()](https://www.javatpoint.com/java-thread-start-method) | It is used to start the execution of the thread. |
| 2) | void | [run()](https://www.javatpoint.com/java-thread-run-method) | It is used to do an action for a thread. |
| 3) | static void | [sleep()](https://www.javatpoint.com/java-thread-sleep-method) | It sleeps a thread for the specified amount of time. |
| 4) | static Thread | [currentThread()](https://www.javatpoint.com/java-thread-currentthread-method) | It returns a reference to the currently executing thread object. |
| 5) | void | [join()](https://www.javatpoint.com/java-thread-join-method) | It waits for a thread to die. |
| 6) | int | [getPriority()](https://www.javatpoint.com/java-thread-getpriority-method) | It returns the priority of the thread. |
| 7) | void | [setPriority()](https://www.javatpoint.com/java-thread-setpriority-method) | It changes the priority of the thread. |
| 8) | String | [getName()](https://www.javatpoint.com/java-thread-getname-method) | It returns the name of the thread. |
| 9) | void | [setName()](https://www.javatpoint.com/java-thread-setname-method) | It changes the name of the thread. |
| 10) | long | [getId()](https://www.javatpoint.com/java-thread-getid-method) | It returns the id of the thread. |
| 11) | boolean | [isAlive()](https://www.javatpoint.com/java-thread-isalive-method) | It tests if the thread is alive. |
| 12) | static void | [yield()](https://www.javatpoint.com/java-thread-yield-method) | It causes the currently executing thread object to pause and allow other threads to execute temporarily. |
| 13) | void | [suspend()](https://www.javatpoint.com/java-thread-suspend-method) | It is used to suspend the thread. |
| 14) | void | [resume()](https://www.javatpoint.com/java-thread-resume-method) | It is used to resume the suspended thread. |
| 15) | void | [stop()](https://www.javatpoint.com/java-thread-stop-method) | It is used to stop the thread. |
| 16) | void | [destroy()](https://www.javatpoint.com/java-thread-destroy-method) | It is used to destroy the thread group and all of its subgroups. |
| 17) | boolean | [isDaemon()](https://www.javatpoint.com/java-thread-isdaemon-method) | It tests if the thread is a daemon thread. |
| 18) | void | [setDaemon()](https://www.javatpoint.com/java-thread-setdaemon-method) | It marks the thread as daemon or user thread. |
| 19) | void | [interrupt()](https://www.javatpoint.com/java-thread-interrupt-method) | It interrupts the thread. |
| 20) | boolean | [isinterrupted()](https://www.javatpoint.com/java-thread-isinterrupted-method) | It tests whether the thread has been interrupted. |
| 21) | static boolean | [interrupted()](https://www.javatpoint.com/java-thread-interrupted-method) | It tests whether the current thread has been interrupted. |
| 22) | static int | [activeCount()](https://www.javatpoint.com/java-thread-activecount-method) | It returns the number of active threads in the current thread's thread group. |
| 23) | void | [checkAccess()](https://www.javatpoint.com/java-thread-checkaccess-method) | It determines if the currently running thread has permission to modify the thread. |
| 24) | static boolean | [holdLock()](https://www.javatpoint.com/java-thread-holdlock-method) | It returns true if and only if the current thread holds the monitor lock on the specified object. |
| 25) | static void | [dumpStack()](https://www.javatpoint.com/java-thread-dumpstack-method) | It is used to print a stack trace of the current thread to the standard error stream. |
| 26) | StackTraceElement[] | [getStackTrace()](https://www.javatpoint.com/java-thread-getstacktrace-method) | It returns an array of stack trace elements representing the stack dump of the thread. |
| 27) | static int | [enumerate()](https://www.javatpoint.com/java-thread-enumerate-method) | It is used to copy every active thread's thread group and its subgroup into the specified array. |
| 28) | Thread.State | [getState()](https://www.javatpoint.com/java-thread-getstate-method) | It is used to return the state of the thread. |
| 29) | ThreadGroup | [getThreadGroup()](https://www.javatpoint.com/java-thread-getthreadgroup-method) | It is used to return the thread group to which this thread belongs |
| 30) | String | [toString()](https://www.javatpoint.com/java-thread-tostring-method) | It is used to return a string representation of this thread, including the thread's name, priority, and thread group. |
| 31) | void | [notify()](https://www.javatpoint.com/java-thread-notify-method) | It is used to give the notification for only one thread which is waiting for a particular object. |
| 32) | void | [notifyAll()](https://www.javatpoint.com/java-thread-notifyall-method) | It is used to give the notification to all waiting threads of a particular object. |
| 33) | void | [setContextClassLoader()](https://www.javatpoint.com/java-thread-setcontextclassloader-method) | It sets the context ClassLoader for the Thread. |
| 34) | ClassLoader | [getContextClassLoader()](https://www.javatpoint.com/java-thread-getcontextclassloader-method) | It returns the context ClassLoader for the thread. |
| 35) | static Thread.UncaughtExceptionHandler | [getDefaultUncaughtExceptionHandler()](https://www.javatpoint.com/java-thread-getdefaultuncaughtexceptionhandler-method) | It returns the default handler invoked when a thread abruptly terminates due to an uncaught exception. |
| 36) | static void | [setDefaultUncaughtExceptionHandler()](https://www.javatpoint.com/java-thread-setdefaultuncaughtexceptionhandler-method) | It sets the default handler invoked when a thread abruptly terminates due to an uncaught exception. |

# Life cycle of a Thread (Thread States)

In Java, a thread always exists in any one of the following states. These states are:

1. New
2. Active
3. Blocked / Waiting
4. Timed Waiting
5. Terminated

## **Explanation of Different Thread States**

**New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.

**Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.

ADVERTISEMENT

* **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.  
  A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
* **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

**Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.

For example, a thread (let's say its name is A) may want to print some data from the printer. However, at the same time, the other thread (let's say its name is B) is using the printer to print some data. Therefore, thread A has to wait for thread B to use the printer. Thus, thread A is in the blocked state. A thread in the blocked state is unable to perform any execution and thus never consume any cycle of the Central Processing Unit (CPU). Hence, we can say that thread A remains idle until the thread scheduler reactivates thread A, which is in the waiting or blocked state.

When the main thread invokes the join() method then, it is said that the main thread is in the waiting state. The main thread then waits for the child threads to complete their tasks. When the child threads complete their job, a notification is sent to the main thread, which again moves the thread from waiting to the active state.

If there are a lot of threads in the waiting or blocked state, then it is the duty of the thread scheduler to determine which thread to choose and which one to reject, and the chosen thread is then given the opportunity to run.

**Timed Waiting:** Sometimes, waiting for leads to starvation. For example, a thread (its name is A) has entered the critical section of a code and is not willing to leave that critical section. In such a scenario, another thread (its name is B) has to wait forever, which leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.

**Terminated:** A thread reaches the termination state because of the following reasons:

* When a thread has finished its job, then it exists or terminates normally.
* **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

A terminated thread means the thread is no more in the system. In other words, the thread is dead, and there is no way one can respawn (active after kill) the dead thread.

* The following diagram shows the different states involved in the life cycle of a thread.



* **Explanation:** Whenever we spawn a new thread, that thread attains the new state. When the method start() is invoked on a thread, the thread scheduler moves that thread to the runnable state. Whenever the join() method is invoked on any thread instance, the current thread executing that statement has to wait for this thread to finish its execution, i.e., move that thread to the terminated state. Therefore, before the final print statement is printed on the console, the program invokes the method join() on thread t2, making the thread t1 wait while the thread t2 finishes its execution and thus, the thread t2 get to the terminated or dead state. Thread t1 goes to the waiting state because it is waiting for thread t2 to finish it's execution as it has invoked the method join() on thread t2.

# Thread Scheduler in Java

A component of Java that decides which thread to run or execute and which thread to wait is called a **thread scheduler in Java**. In Java, a thread is only chosen by a thread scheduler if it is in the runnable state. However, if there is more than one thread in the runnable state, it is up to the thread scheduler to pick one of the threads and ignore the other ones. There are some criteria that decide which thread will execute first. There are two factors for scheduling a thread i.e. **Priority** and **Time of arrival**.

**Priority:** Priority of each thread lies between 1 to 10. If a thread has a higher priority, it means that thread has got a better chance of getting picked up by the thread scheduler.

**Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the other threads.

## **Thread Scheduler Algorithms**

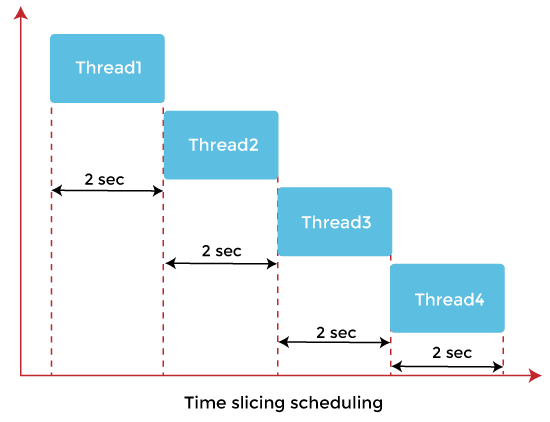
On the basis of the above-mentioned factors, the scheduling algorithm is followed by a Java thread scheduler.

### **Time-slicing scheduling:**

Usually, the First Come First Serve algorithm is non-preemptive, which is bad as it may lead to infinite blocking (also known as starvation). To avoid that, some time-slices are provided to the threads so that after some time, the running thread has to give up the CPU. Thus, the other waiting threads also get time to run their job.

### **Preemptive-Priority Scheduling:**

The name of the scheduling algorithm denotes that the algorithm is related to the priority of the thread



# Thread.sleep() in Java with Examples

The Java Thread class provides the two variant of the sleep() method. First one accepts only an arguments, whereas the other variant accepts two arguments. The method sleep() is being used to halt the working of a thread for a given amount of time. The time up to which the thread remains in the sleeping state is known as the sleeping time of the thread. After the sleeping time is over, the thread starts its execution from where it has left.

### **The sleep() Method Syntax:**

Following are the syntax of the sleep() method.

1. **public** **static** **void** sleep(**long** mls) **throws** InterruptedException
2. **public** **static** **void** sleep(**long** mls, **int** n) **throws** InterruptedException

The method sleep() with the one parameter is the native method, and the implementation of the native method is accomplished in another programming language. The other methods having the two parameters are not the native method. That is, its implementation is accomplished in Java. We can access the sleep() methods with the help of the Thread class, as the signature of the sleep() methods contain the static keyword. The native, as well as the non-native method, throw a checked Exception. Therefore, either try-catch block or the throws keyword can work here.

The Thread.sleep() method can be used with any thread. It means any other thread or the main thread can invoke the sleep() method.

### **Important Points to Remember About the Sleep() Method**

Whenever the Thread.sleep() methods execute, it always halts the execution of the current thread.

Whenever another thread does interruption while the current thread is already in the sleep mode, then the InterruptedException is thrown.

* If the system that is executing the threads is busy, then the actual sleeping time of the thread is generally more as compared to the time passed in arguments. However, if the system executing the sleep() method has less load, then the actual sleeping time of the thread is almost equal to the time passed in the argument.

# Can we start a thread twice

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

* What if we call Java run() method directly instead start() method?
* Each thread starts in a separate call stack.
* Invoking the run() method from the main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack.

# Java join() method

The join() method in Java is provided by the java.lang.Thread class that permits one thread to wait until the other thread to finish its execution. Suppose th be the object the class Thread whose thread is doing its execution currently, then the th.join(); statement ensures that th is finished before the program does the execution of the next statement. When there are more than one thread invoking the join() method, then it leads to overloading on the join() method that permits the developer or programmer to mention the waiting period. However, similar to the sleep() method in Java, the join() method is also dependent on the operating system for the timing, so we should not assume that the join() method waits equal to the time we mention in the parameters. The following are the three overloaded join() methods.

## **Description of The Overloaded join() Method**

**join():** When the join() method is invoked, the current thread stops its execution and the thread goes into the wait state. The current thread remains in the wait state until the thread on which the join() method is invoked has achieved its dead state. If interruption of the thread occurs, then it throws the InterruptedException.

**Syntax:**

* **Iterable Interface**

**The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface.**

**It contains only one abstract method. i.e.,**

**Iterator<T> iterator()**

**It returns the iterator over the elements of type T.**

## **List Interface**

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack.

## **Collection Interface**

The Collection interface is the interface which is implemented by all the classes in the collection framework. It declares the methods that every collection will have. In other words, we can say that the Collection interface builds the foundation on which the collection framework depends.

Some of the methods of Collection interface are Boolean add ( Object obj), Boolean addAll ( Collection c), void clear(), etc. which are implemented by all the subclasses of Collection interface.

## **ArrayList**

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed.

## **LinkedList**

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

## **Vector**

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

## **Stack**

The stack is the subclass of Vector. It implements the last-in-first-out data structure, i.e., Stack. The stack contains all of the methods of Vector class and also provides its methods like boolean push(), boolean peek(), boolean push(object o), which defines its properties.

## **Queue Interface**

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

## **PriorityQueue**

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

## **Deque Interface**

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

## **ArrayDeque**

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

## **Set Interface**

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set. Set is implemented by HashSet, LinkedHashSet, and TreeSet.

## **HashSet**

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

## **LinkedHashSet**

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

## **SortedSet Interface**

SortedSet is the alternate of Set interface that provides a total ordering on its elements. The elements of the SortedSet are arranged in the increasing (ascending) order. The SortedSet provides the additional methods that inhibit the natural ordering of the elements.

## **TreeSet**

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Java Collections class

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

The important points about Java Collections class are:

* Java Collection class supports the **polymorphic algorithms** that operate on collections.
* Java Collection class throws a **NullPointerException** if the collections or class objects provided to them are null.
* Difference between Comparable and Comparator
* Comparable and Comparator both are interfaces and can be used to sort collection elements.
* However, there are many differences between Comparable and Comparator interfaces that are given below.

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

**34 Java Collections Interview Questions**

**In Java, collection interview questions are most asked by the interviewers. Here is the list of the most asked collections interview questions with answers.**

**1) What is the Collection framework in Java?**

**Collection Framework is a combination of classes and interface, which is used to store and manipulate the data in the form of objects. It provides various classes such as ArrayList, Vector, Stack, and HashSet, etc. and interfaces such as List, Queue, Set, etc. for this purpose**

**2) What are the main differences between array and collection?**

**Array and Collection are somewhat similar regarding storing the references of objects and manipulating the data, but they differ in many ways. The main differences between the array and Collection are defined below:**

* **Arrays are always of fixed size, i.e., a user can not increase or decrease the length of the array according to their requirement or at runtime, but In Collection, size can be changed dynamically as per need.**
* **Arrays can only store homogeneous or similar type objects, but in Collection, heterogeneous objects can be stored.**
* **Arrays cannot provide the ?ready-made? methods for user requirements as sorting, searching, etc. but Collection includes readymade methods to use.**

### **4) What is the difference between ArrayList and Vector?**

|  |  |  |
| --- | --- | --- |
| **No.** | **ArrayList** | **Vector** |
| 1) | ArrayList is not synchronized. | Vector is synchronized. |
| 2) | ArrayList is not a legacy class. | Vector is a legacy class. |
| 3) | ArrayList increases its size by 50% of the array size. | Vector increases its size by doubling the array size. |
| 4) | ArrayList is not ?thread-safe? as it is not synchronized. | Vector list is ?thread-safe? as it?s every method is synchronized |

**5) What is the difference between ArrayList and LinkedList?**

|  |  |  |
| --- | --- | --- |
| **No.** | **ArrayList** | **LinkedList** |
| **1)** | **ArrayList uses a dynamic array.** | **LinkedList uses a doubly linked list.** |
| **2)** | **ArrayList is not efficient for manipulation because too much is required.** | **LinkedList is efficient for manipulation.** |
| **3)** | **ArrayList is better to store and fetch data.** | **LinkedList is better to manipulate data.** |
| **4)** | **ArrayList provides random access.** | **LinkedList does not provide random access.** |
| **5)** | **ArrayList takes less memory overhead as it stores only object** | **LinkedList takes more memory overhead, as it stores the object as well as the address of that object.** |

**6) What is the difference between Iterator and ListIterator?**

**Iterator traverses the elements in the forward direction only whereas ListIterator traverses the elements into forward and backward direction.**

|  |  |  |
| --- | --- | --- |
| **No.** | **Iterator** | **ListIterator** |
| **1)** | **The Iterator traverses the elements in the forward direction only.** | **ListIterator traverses the elements in backward and forward directions both.** |
| **2)** | **The Iterator can be used in List, Set, and Queue.** | **ListIterator can be used in List only.** |
| **3)** | **The Iterator can only perform remove operation while traversing the collection.** | **ListIterator can perform ?add,? ?remove,? and ?set? operation while traversing the collection.** |

**7) What is the difference between Iterator and Enumeration?**

|  |  |  |
| --- | --- | --- |
| **No.** | **Iterator** | **Enumeration** |
| **1)** | **The Iterator can traverse legacy and non-legacy elements.** | **Enumeration can traverse only legacy elements.** |
| **2)** | **The Iterator is fail-fast.** | **Enumeration is not fail-fast.** |
| **3)** | **The Iterator is slower than Enumeration.** | **Enumeration is faster than Iterator.** |
| **4)** | **The Iterator can perform remove operation while traversing the collection.** | **The Enumeration can perform only traverse operation on the collection.** |

**8) What is the difference between List and Set?**

**The List and Set both extend the collection interface. However, there are some differences between the both which are listed below.**

* **The List can contain duplicate elements whereas Set includes unique items.**
* **The List is an ordered collection which maintains the insertion order whereas Set is an unordered collection which does not preserve the insertion order.**
* **The List interface contains a single legacy class which is Vector class whereas Set interface does not have any legacy class.**
* **The List interface can allow n number of null values whereas Set interface only allows a single null value.**

### **9) What is the difference between HashSet and TreeSet?**

The HashSet and TreeSet, both classes, implement Set interface. The differences between the both are listed below.

* **HashSet maintains�no order�whereas TreeSet maintains�ascending order.**
* **HashSet impended by hash table whereas TreeSet implemented by a Tree structure.**
* **HashSet performs faster than TreeSet.**
* **HashSet is backed by HashMap whereas TreeSet is backed by TreeMap.**

**10) What is the difference between Set and Map?**

**The differences between the Set and Map are given below.**

* **Set contains values only whereas Map contains key and values both.**
* **Set contains unique values whereas Map can contain unique Keys with duplicate values.**
* **Set holds a single number of null value whereas Map can include a single null key with n number of null values.**

**11) What is the difference between HashSet and HashMap?**

**The differences between the HashSet and HashMap are listed below.**

* **HashSet contains only values whereas HashMap includes the entry (key, value). HashSet can be iterated, but HashMap needs to convert into Set to be iterated.**
* **HashSet implements Set interface whereas HashMap implements the Map interface**
* **HashSet cannot have any duplicate value whereas HashMap can contain duplicate values with unique keys.**
* **HashSet contains the only single number of null value whereas HashMap can hold a single null key with n number of null values.**

**12) What is the difference between HashMap and TreeMap?**

**The differences between the HashMap and TreeMap are given below.**

* **HashMap maintains�no order, but TreeMap maintains�ascending order.**
* **HashMap is implemented by hash table whereas TreeMap is implemented by a Tree structure.**
* **HashMap can be sorted by Key or value whereas TreeMap can be sorted by Key.**
* **HashMap may contain a null key with multiple null values whereas TreeMap cannot hold a null key but can have multiple null values.**

**13) What is the difference between HashMap and Hashtable?**

|  |  |  |
| --- | --- | --- |
| **No.** | **HashMap** | **Hashtable** |
| **1)** | **HashMap is not synchronized.** | **Hashtable is synchronized.** |
| **2)** | **HashMap can contain one null key and multiple null values.** | **Hashtable cannot contain any null key or null value.** |
| **3)** | **HashMap is not ?thread-safe,? so it is useful for non-threaded applications.** | **Hashtable is thread-safe, and it can be shared between various threads.** |
| **4)** | **4) HashMap inherits the AbstractMap class** | **Hashtable inherits the Dictionary class.** |

**14) What is the difference between Collection and Collections?**

**The differences between the Collection and Collections are given below.**

* **The Collection is an interface whereas Collections is a class.**
* **The Collection interface provides the standard functionality of data structure to List, Set, and Queue. However, Collections class is to sort and synchronize the collection elements.**
* **The Collection interface provides the methods that can be used for data structure whereas Collections class provides the static methods which can be used for various operation on a collection.**

**15) What is the difference between Comparable and Comparator?**

|  |  |  |
| --- | --- | --- |
| **No.** | **Comparable** | **Comparator** |
| **1)** | **Comparable provides only one sort of sequence.** | **The Comparator provides multiple sorts of sequences.** |
| **2)** | **It provides one method named compareTo().** | **It provides one method named compare().** |
| **3)** | **It is found in java.lang package.** | **It is located in java.util package.** |
| **4)** | **If we implement the Comparable interface, The actual class is modified.** | **The actual class is not changed.** |

**16) What do you understand by BlockingQueue?**

**BlockingQueue is an interface which extends the Queue interface. It provides concurrency in the operations like retrieval, insertion, deletion. While retrieval of any element, it waits for the queue to be non-empty. While storing the elements, it waits for the available space. BlockingQueue cannot contain null elements, and implementation of BlockingQueue is thread-safe.**

**Syntax:**

1. **public interface BlockingQueue<E> extends Queue <E>**

**17) What is the advantage of Properties file?**

**If you change the value in the properties file, you don't need to recompile the java class. So, it makes the application easy to manage. It is used to store information which is to be changed frequently. Consider the following example.**

**8) What does the hashCode() method?**

**The hashCode() method returns a hash code value (an integer number)**

**The hashCode() method returns the same integer number if two keys (by calling equals() method) are identical.**

**However, it is possible that two hash code numbers can have different or the same keys.**

**If two objects do not produce an equal result by using the equals() method, then the hashcode() method will provide the different integer result for both the objects.**

**19) Why we override equals() method?**

**The equals method is used to check whether two objects are the same or not. It needs to be overridden if we want to check the objects based on the property.**

**For example, Employee is a class that has 3 data members: id, name, and salary. However, we want to check the equality of employee object by the salary. Then, we need to override the equals() method.**

**20) How to synchronize List, Set and Map elements?**

**Yes, Collections class provides methods to make List, Set or Map elements as synchronized:**

**21) What is the advantage of the generic collection?**

**There are three main advantages of using the generic collection.**

* **If we use the generic class, we don't need typecasting.**
* **It is type-safe and checked at compile time.**
* **Generic confirms the stability of the code by making it bug detectable at compile time.**

**22) What is hash-collision in Hashtable and how it is handled in Java?**

**Two different keys with the same hash value are known as hash-collision. Two separate entries will be kept in a single hash bucket to avoid the collision. There are two ways to avoid hash-collision.**

* **Separate Chaining**
* **Open Addressing**

**23) What is the Dictionary class?**

**The Dictionary class provides the capability to store key-value pairs.**

**24) What is the default size of load factor in hashing based collection?**

**The default size of load factor is 0.75. The default capacity is computed as initial capacity \* load factor. For example, 16 \* 0.75 = 12. So, 12 is the default capacity of Map.**

**25) What do you understand by fail-fast?**

**The Iterator in java which immediately throws ConcurrentmodificationException, if any structural modification occurs i**

**in, is called as a Fail-fast iterator. Fail-fats iterator does not require any extra space in memory.** **26)**[**What is the difference between Array and ArrayList?**](https://www.javatpoint.com/array-vs-arraylist-in-java)

**The main differences between the Array and ArrayList are given below.**

|  |  |  |
| --- | --- | --- |
| **SN** | **Array** | **ArrayList** |
| **1** | **The Array is of fixed size, means we cannot resize the array as per need.** | **ArrayList is not of the fixed size we can change the size dynamically.** |
| **2** | **Arrays are of the static type.** | **ArrayList is of dynamic size.** |
| **3** | **Arrays can store primitive data types as well as objects.** | **ArrayList cannot store the primitive data types it can only store the objects.** |

**27)**[**What is the difference between the length of an Array and size of ArrayList?**](https://www.javatpoint.com/difference-between-length-of-array-and-size-of-arraylist-in-java)

**The length of an array can be obtained using the property of length whereas ArrayList does not support length property, but we can use size() method to get the number of objects in the list.**

**Finding the length of the array**

1. **Int [] array = new int[4];**
2. **System.out.println("The size of the array is " + array.length);**

**Finding the size of the ArrayList**

1. **ArrayList<String> list=new ArrayList<String>();**
2. **list.add("ankit");��**
3. **list.add("nippun");**
4. **System.out.println(list.size());**

**28)**[**How to convert ArrayList to Array and Array to ArrayList?**](https://www.javatpoint.com/how-to-convert-arraylist-to-array-and-array-to-arraylist-in-java)

**We can convert an Array to ArrayList by using the asList() method of Arrays class. asList() method is the static method of Arrays class and accepts the List object**

1. **rrays.asList(item)**

**We can convert an ArrayList to Array using toArray() method of the ArrayList class. Consider the following syntax to convert the ArrayList to the List object.**

1. **List\_object.toArray(new�String[List\_object.size()])**

**29)**[**How to make Java ArrayList Read-Only?**](https://www.javatpoint.com/how-to-make-java-arraylist-read-only)

**We can obtain java ArrayList Read-only by calling the Collections.unmodifiableCollection() method. When we define an ArrayList as Read-only then we cannot perform any modification in the collection through �add(), remove() or set() method.**

**30)**[**How to remove duplicates from ArrayList?**](https://www.javatpoint.com/how-to-remove-duplicates-from-arraylist-in-java)

**There are two ways to remove duplicates from the ArrayList.**

* **Using HashSet: By using HashSet we can remove the duplicate element from the ArrayList, but it will not then preserve the insertion order.**
* **Using LinkedHashSet: We can also maintain the insertion order by using LinkedHashSet instead of HashSet.**

**The Process to remove duplicate elements from ArrayList using the LinkedHashSet:**