# Primitive data types

The Java programming language is statically-typed, which means that all variables must first be declared before they can be used. This involves stating the variable's type and name, as you've already seen:

int gear = 1;

Doing so tells your program that a field named "gear" exists, holds numerical data, and has an initial value of "1". A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, the Java programming language supports seven other primitive data types. A primitive type is predefined by the language and is named by a reserved keyword. Primitive values do not share state with other primitive values. The eight primitive data types supported by the Java programming language are:

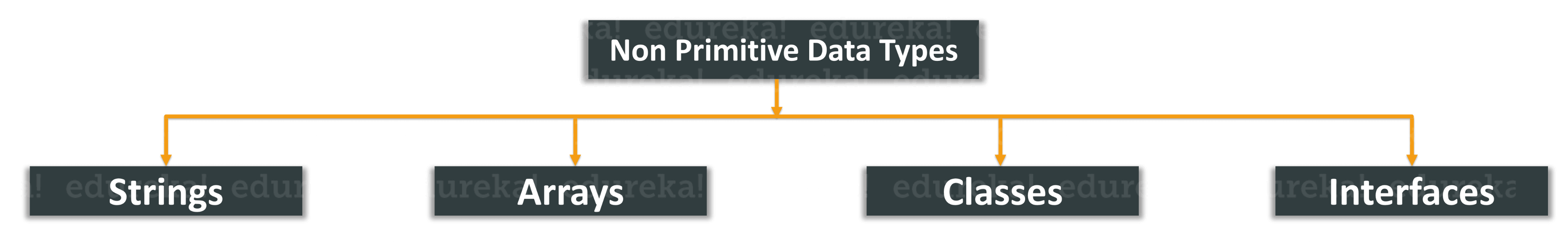
1. **byte**: The byte data type is an 8-bit signed two's complement integer. It has a minimum value of -128 and a maximum value of 127 (inclusive). The byte data type can be useful for saving memory in large arrays, where the memory savings actually matters. They can also be used in place of int where their limits help to clarify your code; the fact that a variable's range is limited can serve as a form of documentation.
2. **short**: The short data type is a 16-bit signed two's complement integer. It has a minimum value of -32,768 and a maximum value of 32,767 (inclusive). As with byte, the same guidelines apply: you can use a short to save memory in large arrays, in situations where the memory savings actually matters.
3. **int**: By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -231 and a maximum value of 231-1. In Java SE 8 and later, you can use the int data type to represent an unsigned 32-bit integer, which has a minimum value of 0 and a maximum value of 232-1. Use the Integer class to use int data type as an unsigned integer. See the section The Number Classes for more information. Static methods like compareUnsigned, divideUnsigned etc have been added to the Integer class to support the arithmetic operations for unsigned integers.
4. **long**: The long data type is a 64-bit two's complement integer. The signed long has a minimum value of -263 and a maximum value of 263-1. In Java SE 8 and later, you can use the long data type to represent an unsigned 64-bit long, which has a minimum value of 0 and a maximum value of 264-1. Use this data type when you need a range of values wider than those provided by int. The Long class also contains methods like compareUnsigned, divideUnsigned etc to support arithmetic operations for unsigned long.
5. **float**: The float data type is a single-precision 32-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. As with the recommendations for byte and short, use a float (instead of double) if you need to save memory in large arrays of floating point numbers. This data type should never be used for precise values, such as currency. For that, you will need to use the java.math.BigDecimal class instead. Numbers and Strings covers BigDecimal and other useful classes provided by the Java

platform.

1. **double**: The double data type is a double-precision 64-bit IEEE 754 floating point. Its range of values is beyond the scope of this discussion, but is specified in the Floating-Point Types, Formats, and Values section of the Java Language Specification. For decimal values, this data type is generally the default choice. As mentioned above, this data type should never be used for precise values, such as currency.
2. **boolean**: The boolean data type has only two possible values: true and false. Use this data type for simple flags that track true/false conditions. This data type represents one bit of information, but its "size" isn't something that's precisely defined.
3. **char**: The char data type is a single 16-bit Unicode character. It has a minimum value of '\u0000' (or 0) and a maximum value of '\uffff' (or 65,535 inclusive).

# Non-Primitive datatypes

Non-Primitive data types refer to objects and hence they are called **reference types.**Examples of non-primitive types include Strings, Arrays, Classes, Interface, etc. Below image depicts various non-primitive data types.



Let’s now understand these non-primitive data types in short.

**Strings:** String is a sequence of characters. But in Java, a string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object. If you wish to know more about Java Strings, you can refer to this article on [Strings in Java](https://www.edureka.co/blog/java-string/).

**Arrays:** Arrays in Java are homogeneous data structures implemented in Java as objects. Arrays store one or more values of a specific data type and provide indexed access to store the same. A specific element in an array is accessed by its index. If you wish to learn Arrays in detail, then kindly check out this article on [Java Arrays](https://www.edureka.co/blog/java-array/).

**Classes:**A [class in Java](https://www.edureka.co/blog/java-tutorial/#obj) is a blueprint which includes all your data.  A class contains fields(variables) and methods to describe the behavior of an object.

**Interface:** Like a class, an interface can have methods and variables, but the methods declared in [interface](https://www.edureka.co/blog/java-collections/#interface) are by default abstract (only method signature, no body).

So that was all about the non-primitive data types. Now let’s understand the difference between primitive and non-primitive data types.

## Difference between primitive and non-primitive data types

The  difference between **primitive** and **non-primitive** data types are as follows:

* Primitive types are predefined in [Java](https://www.edureka.co/blog/what-is-java/). Non-primitive types are created by the programmer and is not defined by Java.
* Non Primitive types can be used to call methods to perform certain operations, while primitive types cannot.
* A primitive type always has a value, whereas non-primitive types can be null.
* A primitive type starts with a lowercase letter, while non-primitive types start with an uppercase letter.
* The size of a primitive type depends on the data type, while non-primitive types have all the same size.

# Java Tokens

In Java, the program contains classes and methods. Further, the methods contain the expressions and statements required to perform a specific operation. These statements and expressions are made up of tokens. In other words, we can say that the expression and statement is a set of tokens. The tokens are the small building blocks of a Java program that are meaningful to the [Java](https://www.javatpoint.com/java-tutorial) compiler. Further, these two components contain variables, constants, and operators. In this section, we will discuss what is tokens in Java.

**What is token in Java?**

The Java compiler breaks the line of code into text (words) is called Java tokens. These are the smallest element of the [Java program](https://www.javatpoint.com/java-programs). The Java compiler identified these words as tokens. These tokens are separated by the delimiters. It is useful for compilers to detect errors. Remember that the delimiters are not part of the Java tokens.

***token <= identifier | keyword | separator | operator | literal | comment***

## **Types of Tokens - Java token includes the following:**

* Keywords
* Identifiers
* Literals
* Operators
* Separators
* Comments

Keywords: These are the pre-defined reserved words of any programming language. Each [keyword](https://www.javatpoint.com/java-keywords) has a special meaning. It is always written in lower case. Java provides the following keywords:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| abstract | boolean | byte | break | class | native | new | package | private | protected |
| case | catch | char | continue | default | public | return | short | static | super |
| do | double | else | extends | final | switch | synchronized | this | thro | throws |
| finally | float | for | if | implements | transient | try | void | volatile | while |
| import | instanceof | int | interface | long | assert | const | enum | goto | strictfp |

Identifier: Identifiers are used to name a variable, constant, function, class, and array. It usually defined by the user. It uses letters, underscores, or a dollar sign as the first character. The label is also known as a special kind of identifier that is used in the goto statement. Remember that the identifier name must be different from the reserved keywords. There are some rules to declare identifiers are:

* The first letter of an identifier must be a letter, underscore or a dollar sign. It cannot start with digits but may contain digits.
* The whitespace cannot be included in the identifier.
* Identifiers are case sensitive.

**Literals:** In programming literal is a notation that represents a fixed value (constant) in the source code. It can be categorized as an integer literal, string literal, Boolean literal, etc. It is defined by the programmer. Once it has been defined cannot be changed. Java provides five types of literals are as follows:

* Integer
* Floating Point
* Character
* String
* Boolean

|  |  |
| --- | --- |
| **Literal** | **Type** |
| 23 | int |
| 9.86 | double |
| false, true | boolean |
| 'K', '7', '-' | char |
| "Chaitanya" | String |
| null | any reference type |

**Operators:** In programming, operators are the special symbol that tells the compiler to perform a special operation. Java provides different types of operators that can be classified according to the functionality they provide. There are eight types of [operators in Java](https://www.javatpoint.com/operators-in-java), are as follows:

* Arithmetic Operators
* Assignment Operators
* Relational Operators
* Unary Operators
* Logical Operators
* Ternary Operators
* Bitwise Operators
* Shift Operators

|  |  |
| --- | --- |
| **Operator** | **Symbols** |
| Arithmetic | + , - , / , \* , % |
| Unary | ++ , - - , ! |
| Assignment | = , += , -= , \*= , /= , %= , ^= |
| Relational | ==, != , < , >, <= , >= |
| Logical | && , || |
| Ternary | (Condition) ? (Statement1) : (Statement2); |
| Bitwise | & , | , ^ , ~ |
| Shift | << , >> , >>> |

**Separators:** The separators in Java is also known as **punctuators**. There are nine separators in Java, are as follows:

separator <= ; | , | . | ( | ) | { | } | [ | ]

### Note that the first three separators (; , and .) are tokens that separate other tokens, and the last six (3 pairs of braces) separators are also known as delimiters. For example, Math.pow(9, 3); contains nine tokens.

* **Square Brackets []:** It is used to define array elements. A pair of square brackets represents the single-dimensional array, two pairs of square brackets represent the two-dimensional array.
* **Parentheses ():** It is used to call the functions and parsing the parameters.
* **Curly Braces {}:** The curly braces denote the starting and ending of a code block.
* **Comma (,):** It is used to separate two values, statements, and parameters.
* **Assignment Operator (=):** It is used to assign a variable and constant.
* **Semicolon (;):** It is the symbol that can be found at end of the statements. It separates the two statements.
* **Period (.):** It separates the package name form the sub-packages and class. It also separates a variable or method from a reference variable.

**Comments:** [Comments](https://www.javatpoint.com/java-comments) allow us to specify information about the program inside our Java code. Java compiler recognizes these comments as tokens but excludes it form further processing. The Java compiler treats comments as whitespaces. Java provides the following two types of comments:

* **Line Oriented:** It begins with a pair of forwarding slashes (**//**).
* **Block-Oriented:** It begins with /\* and continues until it founds **\*/**.

## **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.

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 dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.

## **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.

## **Static in java**

 Static is used for memory management in java, when something is declared with static keyword it means and memory allocation is done only once for that element. Static keyword can be used for variables, method, blocks and class.

## **Static as Variable**

“static” can be used with variable, it means that variable belongs to class, Java static property is shared to all objects.

* The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
* The static variable gets memory only once in the class area at the time of class loading.
* It makes your program **memory efficient** (i.e., it saves memory).

## **Static method**

If you apply static keyword with any method, it is known as static method.

* A static method belongs to the class rather than the object of a class.
* A static method can be invoked without the need for creating an instance of a class.
* A static method can access static data member and can change the value of it.
* Static methods cannot be overridden, since they are resolved using static binding by the compiler at compile time. However, we can have the same name methods declared static in both superclass and subclass, but it will be called Method Hiding as the derived class method will hide the base class method.
* The static method cannot use non static data member or call non-static method directly.
* this and super cannot be used in static context.

**When to use static methods?**

* When you have code that can be shared across all instances of the same class, put that portion of code into static method.
* They are basically used to access static field(s) of the class.

**Instance Method-** Instance method are methods which require an object of its class to be created before it can be called. To invoke a instance method, we have to create an Object of the class in which the method is defined.

**Memory allocation:** These methods themselves are stored in Permanent Generation space of heap but the parameters (arguments passed to them) and their local variables and the value to be returned are allocated in stack. They can be called within the same class in which they reside or from the different classes defined either in the same package or other packages depend on the access type provided to the desired instance method.

**Important Points:** Instance method(s) belong to the Object of the class, not to the class i.e. they can be called after creating the Object of the class.

Instance methods are not stored on a per-instance basis, even with virtual methods. They’re stored in a single memory location, and they only “know” which object they belong to because this pointer is passed when you call them.

They can be overridden since they are resolved using dynamic binding at run time.

**Instance method vs Static method**

* Instance method can access the instance methods and instance variables directly.
* Instance method can access static variables and static methods directly.
* Static methods can access the static variables and static methods directly.
* Static methods can’t access instance methods and instance variables directly. They must use reference to object. And static method can’t use this keyword as there is no instance for ‘this’ to refer to.

### **Why is the Java main method static?**

Ans) It is because the object is not required to call a static method. If it were a non-static method, [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) creates an object first then call main() method that will lead the problem of extra memory allocation.

## **static block**

* Is used to initialize the static data member.
* It is executed before the main method at the time of classloading.

### **Q) Can we execute a program without main() method?**

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the [main method](https://www.javatpoint.com/java-main-method).

## **this keyword in Java**

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.

**Usage of Java this keyword**

Here is given the 6 usage of java this keyword.

1. **this can be used to refer current class instance variable.-** This keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
2. **this can be used to invoke current class method (implicitly)-** You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.
3. **this() can be used to invoke current class constructor.-** The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.
4. **this can be passed as an argument in the method call.-** This keyword can also be passed as an argument in the method. It is mainly used in the event handling. Application of this that can be passed as an argument:- In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.
5. **this can be passed as argument in the constructor call.-** We can pass this keyword in the constructor also. It is useful if we have to use one object in multiple classes.
6. **this can be used to return the current class instance from the method.-** We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive

# Object oriented programming concepts

## **Inheritance**

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviour of a parent object. It is an important part of [OOPs](https://www.javatpoint.com/java-oops-concepts) (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new [classes](https://www.javatpoint.com/object-and-class-in-java) that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

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

### **Why use inheritance in java**

* For [Method Overriding](https://www.javatpoint.com/method-overriding-in-java) (so [runtime polymorphism](https://www.javatpoint.com/runtime-polymorphism-in-java) can be achieved).
* For Code Reusability.

### **Terms used in Inheritance**

* **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
* **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
* **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
* **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

### **Types of inheritance in java**

**1. Single inheritance-** When a class inherits another class, it is known as a single inheritance.

**2. Multilevel inheritance-** When there is a chain of inheritance, it is known as multilevel inheritance.

**3. Hierarchical Inheritance -** When two or more classes inherits a single class, it is known as hierarchical inheritance

Q) Why multiple inheritance is not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class. Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

## **Aggregation in Java**

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

### **Why use Aggregation?**

* For Code Reusability.

### **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.

## **Polymorphism**

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

**Real-life Illustration: Polymorphism -** A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.

Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word “poly” means many and “morphs” means forms, So it means many forms.

**Types of polymorphism**

* 1. Compile-time Polymorphism
  2. Runtime Polymorphism

**Type 1: Compile-time polymorphism -**It is also known as static polymorphism. This type of polymorphism is achieved by **function overloading or operator overloading.**

Note: But Java doesn’t support the Operator Overloading.

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

**Type 2: Runtime polymorphism**- It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by **Method Overriding**. Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

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

### **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). Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

#### **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

##### **In Java, Method Overloading is not possible by changing the return type of the method only.**

##### 1) Method Overloading: changing no. of arguments

##### 2) Method Overloading: changing data type of arguments

##### **Q) Why Method Overloading is not possible by changing the return type of method only?**

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

System.out.println(Adder.add(11,11)); //Here, how can java determine which sum() method should be called?

###### **Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.**

##### **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. Let's see the simple example:

#### **Method Overloading and Type Promotion**

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:

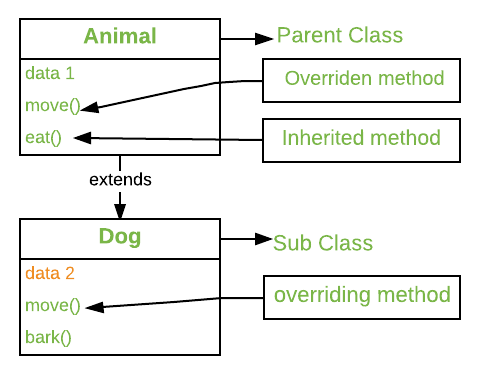


As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.

##### **One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.**

### **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.

Method overriding is one of the way by which java achieve Run Time Polymorphism. The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

#### **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. Overriding and Access-Modifiers : The access modifier for an overriding method can allow more, but not less, access than the overridden method. For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile-time error.
2. Final methods cannot be overridden : If we don’t want a method to be overridden, we declare it as final. Please see Using final with Inheritance
3. Static methods can not be overridden(Method Overriding vs Method Hiding) : When you define a static method with same signature as a static method in base class, it is known as method hiding.
4. Private methods can not be overridden : Private methods cannot be overridden as they are bonded during compile time. Therefore we can’t even override private methods in a subclass.(See this for details).
5. The overriding method must have same return type (or subtype) : From Java 5.0 onwards it is possible to have different return type for a overriding method in child class, but child’s return type should be sub-type of parent’s return type. This phenomena is known as covariant return type.
6. Invoking overridden method from sub-class : We can call parent class method in overriding method using super keyword.
7. Overriding and constructor : We cannot override constructor as parent and child class can never have constructor with same name(Constructor name must always be same as Class name).
8. Overriding and Exception-Handling : Below are two rules to note when overriding methods related to exception-handling.

* Rule#1 : If the super-class overridden method does not throw an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.
* Rule#2 : If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error. Also there is no issue if subclass overridden method is not throwing any exception.

1. Overriding and abstract method: Abstract methods in an interface or abstract class are meant to be overridden in derived concrete classes otherwise a compile-time error will be thrown.
2. Overriding and synchronized/strictfp method : The presence of synchronized/strictfp modifier with method have no effect on the rules of overriding, i.e. it’s possible that a synchronized/strictfp method can override a non-synchronized/strictfp one and vice-versa.

#### **Why Method Overriding ?**

As stated earlier, overridden methods allow Java to support run-time polymorphism. Polymorphism is essential to object-oriented programming for one reason: it allows a general class to specify methods that will be common to all of its derivatives while allowing subclasses to define the specific implementation of some or all of those methods. Overridden methods are another way that Java implements the “one interface, multiple methods” aspect of polymorphism.

Dynamic Method Dispatch is one of the most powerful mechanisms that object-oriented design brings to bear on code reuse and robustness. The ability to exist code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Overridden methods allow us to call methods of any of the derived classes without even knowing the type of derived class object.

#### **When to apply Method Overriding ?(with example)**

Overriding and Inheritance : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.

#### **Difference between method overloading and method overriding in java**

|  |  |
| --- | --- |
| **Method Overloading** | **Method Overriding** |
| Method overloading is used *to increase the readability* of the program. | Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class. |
| Method overloading is performed *within class*. | Method overriding occurs *in two classes* that have IS-A (inheritance) relationship. |
| In case of method overloading, *parameter must be different*. | In case of method overriding, *parameter must be same*. |
| Method overloading is the example of *compile time polymorphism*. | Method overriding is the example of *run time polymorphism*. |
| In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter. | Return type must be same or covariant in method overriding. |

#### **Dynamic Method Dispatch or Runtime Polymorphism in Java**

***Prerequisite: Overriding in java, Inheritance***

Method overriding is one of the ways in which Java supports Runtime Polymorphism. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time.

When an overridden method is called through a superclass reference, Java determines which version(superclass/subclasses) of that method is to be executed based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time.

At run-time, it depends on the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed

A superclass reference variable can refer to a subclass object. This is also known as upcasting. Java uses this fact to resolve calls to overridden me

**In Java, we can override methods only, not the variables(data members), so runtime polymorphism cannot be achieved by data members.**

**Advantages of Dynamic Method Dispatch**

Dynamic method dispatch allow Java to support overriding of methods which is central for run-time polymorphism.

It allows a class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.

It also allow subclasses to add its specific methods subclasses to define the specific implementation of some.

### **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. *Note: If you are beginner to java, skip this topic and return to it after OOPs concepts.*

#### **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.

Let's take an example to understand the advantages of the covariant return type.

### **Super Keyword in Java**

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

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

**Usage of Java super Keyword-**

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

### **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. | |
| There are three places in java where you can perform operations: Method,constructor,block |
| it seems that instance initializer block is firstly invoked but NO. Instance intializer block is invoked at the time of object creation. The java compiler copies the instance initializer block in the constructor after the first statement super(). So firstly, constructor is invoked. Let's understand it by the figure given below: | |

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: |

The instance initializer block is created when instance of the class is created.

The instance initializer block is invoked after the parent class constructor is invoked (i.e. after super() constructor call).

The instance initializer block comes in the order in which they appear.

### final keyword in java**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.

*Q) Is final method inherited?*

Ans) Yes, final method is inherited but you cannot override it. For Example:

*Q) What is blank or uninitialized final variable?*

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

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

It can be initialized only in constructor.

*Que) Can we initialize blank final variable?*

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.

### **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.

class Animal {  
 void eat() {  
 System.*out*.println("animal is eating...");  
 }  
}  
class Dog extends Animal {  
 void eat() {  
 System.*out*.println("dog is eating...");  
 }  
 public static void main(String args[]) {  
 Animal a = new Dog();  
 a.eat();  
 }  
}

## **Abstraction**

Data Abstraction is the property by virtue of which only the essential details are displayed to the user. The trivial or the non-essentials units are not displayed to the user. Ex: A car is viewed as a car rather than its individual components.

*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.*

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of a car or applying brakes will stop the car, but he does not know about how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

In java, abstraction is achieved by interfaces and abstract classes. We can achieve 100% abstraction using interfaces.

### **Abstract class**

#### **Abstract classes and Abstract methods :**

* An abstract class is a class that is declared with an abstract keyword.
* An abstract method is a method that is declared without implementation.
* An abstract class may or may not have all abstract methods. Some of them can be concrete methods
* A method defined abstract must always be redefined in the subclass, thus making overriding compulsory OR either make the subclass itself abstract.
* Any class that contains one or more abstract methods must also be declared with an abstract keyword.
* There can be no object of an abstract class. That is, an abstract class cannot be directly instantiated with the new operator.
* An abstract class can have parameterized constructors and the default constructor is always present in an abstract class.

#### **Abstract Class in java-** Following are some important observations about abstract classes in Java.

* An instance of an abstract class cannot be created.
* Constructors are allowed.
* We can have an abstract class without any abstract method.
* There can be final method in abstract class but any abstract method in class(abstract class) can not be declared as final or in simper terms final method cannot be abstract itself as it will yield error: “Illegal combination of modifiers: abstract and final”
* We are not allowed to create object for any abstract class.
* We can define static methods in an abstract class
* We can use abstract keyword for declaring top level classes (Outer class) as well as inner classes as abstract
* If a class contain at least one abstract method then compulsory we should declare class as abstract
* If Child class is unable to provide implementation to all abstract methods of Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to remaining abstract method

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

Points to Remember-

* An abstract class must be declared with an abstract keyword.
* It can have abstract and non-abstract methods.
* It cannot be instantiated.
* It can have constructors and static methods also.
* It can have final methods which will force the subclass not to change the body of the method.

#### **Abstract class having constructor, data member and methods**

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

* **Rule: If there is an abstract method in a class, that class must be abstract.**
* **Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.**

#### **Few Observations about abstract class-**

1. Observation 1: instance of an abstract class cannot be created; we can have references to abstract class type though.
2. Observation 2: abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of an inherited class is created.
3. Observation 3: In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated but can only be inherited.
4. Observation 4: Abstract classes can also have final methods (methods that cannot be overridden)
5. Observation 5: For any abstract java class we are not allowed to create an object i.e., for abstract class instantiation is not possible.
6. Observation 6: Similar to the interface we can define static methods in an abstract class that can be called independently without an object.
7. Observation 7: We can use abstract keyword for declaring top level classes (Outer class) as well as inner classes as abstract
8. Observation 8: If a class contain at least one abstract method then compulsory we should declare class as abstract otherwise we will get compile time error because If a class contains at least one abstract method then implementation is not complete for that class and hence it is not recommended to create a object so in order to restrict object creation for such partial classes we use abstract keyword
9. Observation 9: If Child class is unable to provide implementation to all abstract methods of Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to remaining abstract method

### **Interface in Java**

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

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

* Java Interface also represents the IS-A relationship.
* It cannot be instantiated just like the abstract class.
* Since Java 8, we can have default and static methods in an interface.
* Since Java 9, we can have private methods in an interface.

#### **Why use Java interface?**

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

* It is used to achieve abstraction, It is used to achieve total abstraction.
* It can be used to achieve loose coupling.
* Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
* Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

The reason is, abstract classes may contain non-final variables, whereas variables in the interface are final, public and static.

// A simple interface

interface Player

{

final int id = 10;

int move();

}

#### **Internal addition by the compiler**

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

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

#### **Advantages of Interfaces in Java**

* Without bothering about the implementation part, we can achieve the security of the implementation.
* In Java, multiple inheritance is not allowed, however, you can use an interface to make use of it as you can implement more than one interface.

#### **Important Points About Interface:**

* We can’t create an instance(interface can’t be instantiated) of the interface but we can make the reference of it that refers to the Object of its implementing class.
* A class can implement more than one interface.
* An interface can extend to another interface or interface (more than one interface).
* A class that implements the interface must implement all the methods in the interface.
* All the methods are public and abstract. And all the fields are public, static, and final.
* It is used to achieve multiple inheritances.
* It is used to achieve loose coupling.
* New Features Added in Interfaces in JDK 9

#### **From Java 9 onwards, interfaces can contain the following also:**

* Static methods
* Private methods
* Private Static methods

#### **What is marker or tagged interface?**

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

#### **Difference between abstract class and interface**

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

**Abstract class vs Interface**

**Final Variables**: Variables declared in a Java interface are by default final. An abstract class may contain non-final variables.

**Type of variables**: Abstract class can have final, non-final, static and non-static variables. The interface has only static and final variables.

**Implementation**: Abstract class can provide the implementation of the interface. Interface can’t provide the implementation of an abstract class.

**Inheritance vs Abstraction**: A Java interface can be implemented using the keyword “implements” and an abstract class can be extended using the keyword “extends”.

**Multiple implementations**: An interface can extend another Java interface only, an abstract class can extend another Java class and implement multiple Java interfaces.

**Accessibility of Data Members**: Members of a Java interface are public by default. A Java abstract class can have class members like private, protected, etc.

## **Encapsulation**

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, 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

**Advantages of Encapsulation:**

* 1. **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.
  2. **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. 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 as write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
  3. **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
  4. **Testing code is easy:** Encapsulated code is easy to test for unit testing.

### **Access Modifiers in Java**

There are two types of modifiers in Java: access modifiers and non-access modifiers. The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

**There are four types of Java access modifiers:**

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 |

1. **Private**: 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 cannot be declared as private because

* private means “only visible within the enclosing class”.
* protected means “only visible within the enclosing class and any subclasses”
* Note: A class cannot be private or protected except nested class.

### Role of Private Constructor- If you make any class constructor private, you cannot create the instance of that class from outside the class.

1. **Default:** 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, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible only within the same package.
2. **Private:** The private access modifier is accessible only within the class.
3. **Public:**

* 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.

## **OOPs Miscellaneous**

### **Object class in Java**

* Object class is present in java.lang package.
* Every class in Java is directly or indirectly derived from the Object class.
* If a class does not extend any other class, then it is a direct child class of Object and if extends another class then it is indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program
* The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, known as upcasting.

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final Class getClass() | returns the Class class object of this object. The Class class can further be used to get the metadata of this class. |
| public int hashCode() | returns the hashcode number for this object. |
| public boolean equals(Object obj) | compares the given object to this object. |
| protected Object clone() throws CloneNotSupportedException | creates and returns the exact copy (clone) of this object. |
| public String toString() | returns the string representation of this object. |
| public final void notify() | wakes up single thread, waiting on this object's monitor. |
| public final void notifyAll() | wakes up all the threads, waiting on this object's monitor. |
| public final void wait(long timeout)throws InterruptedException | causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait(long timeout,int nanos)throws InterruptedException | causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method). |
| public final void wait()throws InterruptedException | causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method). |
| protected void finalize()throws Throwable | is invoked by the garbage collector before object is being garbage collected. |

### **Object Cloning in Java**

* The object cloning is a way to create exact copy of an object. The clone() method of Object class is used to clone an object.
* The java.lang.Cloneable interface must be implemented by the class whose object clone we want to create. If we don't implement Cloneable interface, clone() method generates CloneNotSupportedException.
* The clone() method is defined in the Object class. Syntax of the clone() method is as follows:

*protected Object clone() throws CloneNotSupportedException*

**Why use clone() method ?**

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

**Advantage of Object cloning**

Although Object.clone() has some design issues but it is still a popular and easy way of copying objects. Following is a list of advantages of using clone() method:

* You don't need to write lengthy and repetitive codes. Just use an abstract class with a 4- or 5-line long clone() method.
* It is the easiest and most efficient way for copying objects, especially if we are applying it to an already developed or an old project. Just define a parent class, implement Cloneable in it, provide the definition of the clone() method and the task will be done.
* Clone() is the fastest way to copy array.

**Disadvantage of Object cloning**

Following is a list of some disadvantages of clone() method:

* To use the Object.clone() method, we have to change a lot of syntaxes to our code, like implementing a Cloneable interface, defining the clone() method and handling CloneNotSupportedException, and finally, calling Object.clone() etc.
* We have to implement cloneable interface while it doesn't have any methods in it. We just have to use it to tell the JVM that we can perform clone() on our object.
* Object.clone() is protected, so we have to provide our own clone() and indirectly call Object.clone() from it.
* Object.clone() doesn't invoke any constructor so we don't have any control over object construction.
* If you want to write a clone method in a child class then all of its superclasses should define the clone() method in them or inherit it from another parent class. Otherwise, the super.clone() chain will fail.
* Object.clone() supports only shallow copying but we will need to override it if we need deep cloning.

If we create another object by new keyword and assign the values of another object to this one, it will require a lot of processing on this object. So to save the extra processing task we use clone() method.

**Wrapper classes in Java**

The wrapper class in Java provides the mechanism to convert primitive into object and object into primitive.Since J2SE 5.0, autoboxing and unboxing feature convert primitives into objects and objects into primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

**Use of Wrapper classes in Java-**Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

* Change the value in Method: Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
* Serialization: We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
* Synchronization: Java synchronization works with objects in Multithreading.
* java.util package: The java.util package provides the utility classes to deal with objects.

|  |  |
| --- | --- |
| Primitive Type | Wrapper class |
| boolean | [Boolean](https://www.javatpoint.com/java-boolean) |
| char | [Character](https://www.javatpoint.com/post/java-character) |
| byte | [Byte](https://www.javatpoint.com/java-byte) |
| short | [Short](https://www.javatpoint.com/java-short) |
| int | [Integer](https://www.javatpoint.com/java-integer) |
| long | [Long](https://www.javatpoint.com/java-long) |
| float | [Float](https://www.javatpoint.com/java-float) |
| double | [Double](https://www.javatpoint.com/java-double) |

* Collection Framework: Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

### **Autoboxing**

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.

### **Unboxing**

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing. Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

### **Custom Wrapper class in Java**

Java Wrapper classes wrap the primitive data types, that is why it is known as wrapper classes. We can also create a class which wraps a primitive data type. So, we can create a custom wrapper class in Java.

### **Difference between object and class**

|  |  |
| --- | --- |
| **Object** | **Class** |
| Object is an **instance** of a class. | Class is a **blueprint or template** from which objects are created. |
| Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | Class is a **group of similar objects**. |
| Object is a **physical** entity. | Class is a **logical** entity. |
| Object is created through **new keyword** mainly e.g. | Class is declared using **class keyword** e.g. |
| Student s1=new Student(); | class Student{} |
| Object is created **many times** as per requirement. | Class is declared **once**. |
| Object **allocates memory when it is created**. | Class **doesn't allocated memory when it is created**. |
| There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | There is only **one way to define class** in java using class keyword. |

# Java String---

# Exception Handling in Java

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

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

**Major reasons why an exception Occurs**

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

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

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

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

**Exception**: Exception indicates conditions that a reasonable application might try to catch.

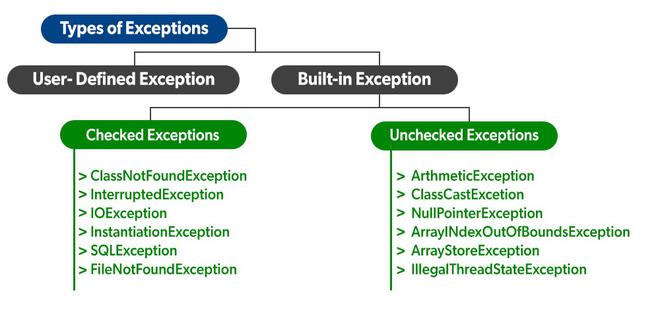
## **Exception Hierarchy**

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



## **Types of Exceptions**

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



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

1. **Built-in Exceptions**

* Checked Exception
* Unchecked Exception

1. **User-Defined Exceptions**

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

**Checked Exceptions**:

* Checked exceptions are called compile-time exceptions because these exceptions are checked at compile-time by the compiler.
* 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.
* A checked exception is an exception that is checked (notified) by the compiler at compilation-time, these are also called as compile time exceptions. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

**Unchecked Exceptions**:

* The unchecked exceptions are just opposite to the checked exceptions. The compiler will not check these exceptions at compile time. In simple words, if a program throws an unchecked exception, and even if we didn’t handle or declare it, the program would not give a compilation error.
* 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.
* An unchecked exception is an exception that occurs at the time of execution. These are also called as Runtime Exceptions. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

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

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

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

|  |  |
| --- | --- |
| 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. |

## **How Does JVM handle an Exception?**

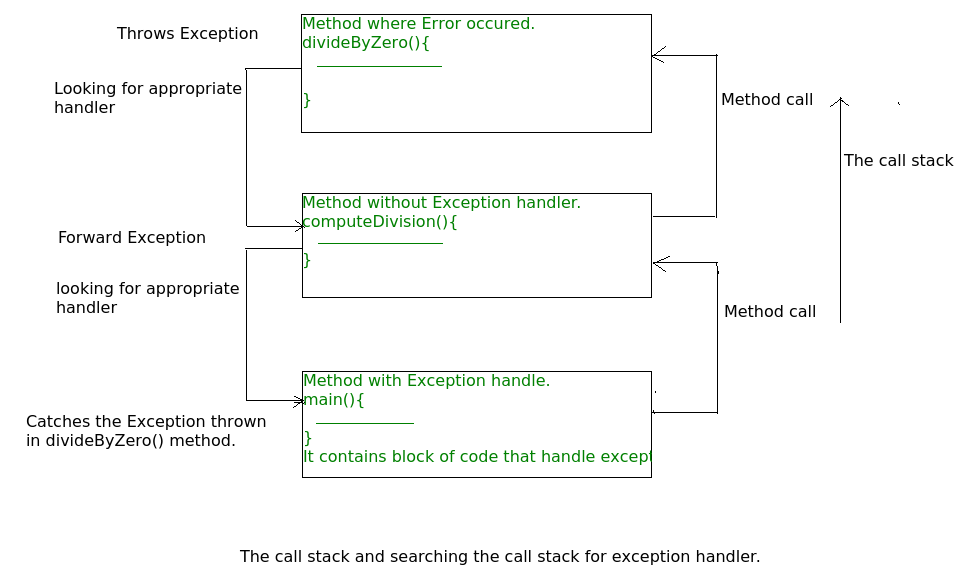
Whenever inside a method, if an exception has occurred, the method creates an Object known as an Exception Object and hands it off to the run-time system(JVM). The exception object contains the name and description of the exception and the current state of the program where the exception has occurred.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

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

*Exception in thread "xxx" Name of Exception : Description*

*... ...... .. // Call Stack*

Look at the below diagram to understand the flow of the call stack.



**How Programmer Handles an Exception?**

**Customized Exception Handling:** Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Briefly, here is how they work. Program statements that you think can raise exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using catch block) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed after a try block completes is put in a finally block.

**Certain below key points are needed to be remembered that are as follows:**

* In a method, there can be more than one statement that might throw an exception, So put all these statements within their own try block and provide a separate exception handler within their own catch block for each of them.
* If an exception occurs within the try block, that exception is handled by the exception handler associated with it. To associate the exception handler, we must put a catch block after it. There can be more than one exception handlers. Each catch block is an exception handler that handles the exception of the type indicated by its argument. The argument, ExceptionType declares the type of exception that it can handle and must be the name of the class that inherits from the Throwable class.
* For each try block there can be zero or more catch blocks, but only one final block.
* The finally block is optional. It always gets executed whether an exception occurred in try block or not. If an exception occurs, then it will be executed after try and catch blocks. And if an exception does not occur then it will be executed after the try block. The finally block in java is used to put important codes such as clean up code e.g. closing the file or closing the connection.

## **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**

**try**{

//code that may throw an exception

}**catch**(Exception\_class\_Name ref){}

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

**try**{

//code that may throw an exception

}**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.

##### **Flowchart of Multi-catch Block**



## **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.

## **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.

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

### **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.

#### **Usage of Java finally**

##### Case 1: When an exception does not occur

* Case 2: When an exception occur but not handled by the catch block
* Case 3: When an exception occurs and is handled by the catch block

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

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

## **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.

### **Java throw keyword Example**

#### **Throwing Unchecked Exception**

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

#### **Throwing Checked Exception**

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

#### **Throwing User-defined Exception**

exception is everything else under the Throwable 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.

throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

In a program, if there is a chance of raising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error saying unreported exception XXX must be caught or declared to be thrown. To prevent this compile time error we can handle the exception in two ways:

* 1. By using try catch
  2. By using throws keyword

We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM) then caller method is responsible to handle that exception.

**Important points to remember about throws keyword:** throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless. throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program. By the help of throws keyword we can provide information to the caller of the method about the exception.

**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.

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

## **Difference between throw and throws in Java**

|  |  |  |
| --- | --- | --- |
| Basis of Differences | throw | throws |
| 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. |
| 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. |  |
| 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. |
| Declaration | throw is used within the method. | throws is used with the method signature. |
| 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**

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

## **Exception Handling with Method Overriding in Java**

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

Some of the rules are listed below:

* **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.**

### **If the superclass method declares an exception**

#### **Rule 1: If the superclass method declares an exception, subclass overridden method can declare the same subclass exception or no exception but cannot declare parent exception.**

## **Java Custom Exception/User defined Exceptions**

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.

An exception is an issue (run time error) that occurred during the execution of a program. When an exception occurred the program gets terminated abruptly and, the code past the line that generated the exception never gets executed.

Java provides us the facility to create our own exceptions which are basically derived classes of Exception. Creating our own Exception is known as a custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user needs. In simple words, we can say that a User-Defined Exception or custom exception is creating your own exception class and throwing that exception using the ‘throw’ keyword.

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.Following are few of the reasons to use 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.

In order to create custom exception, we need to extend Exception class that belongs to java.lang package.

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

Eg-

class InvalidAge extends Exception {  
 InvalidAge(String msg) {  
 super(msg);  
 }  
}  
public class Custom\_Exception {  
 public void validateAge(int age) throws InvalidAge {  
 if(age < 18) {  
 throw new InvalidAge("Age is valid");  
 }  
 }  
 public static void main(String[] args) {  
 Custom\_Exception custom\_exception = new Custom\_Exception();  
 try {  
 custom\_exception.validateAge(16);  
 } catch(Exception e) {  
 e.printStackTrace();  
 }  
 }  
}

# Java Inner class and Interface

## 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.

### **Syntax of Inner class**

**class** Java\_Outer\_class{

 //code

**class** Java\_Inner\_class{

  //code

 }

}

### **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.

#### **Difference between nested class and inner class in Java**

An inner class is a part of a nested class. Non-static nested classes are known as inner classes.

#### **Types of Nested classes**

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

* Non-static nested class (inner class)
  1. Member inner class
  2. Anonymous inner class
  3. Local inner class
* Static nested class

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

### Java Member Inner class

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

**Syntax:**

**class** Outer{

 //code

**class** Inner{

  //code

 }  }

**How to instantiate Member Inner class in Java?**

An object or instance of a member's inner class always exists within an object of its outer class. The new operator is used to create the object of member inner class with slightly different syntax.The general form of syntax to create an object of the member inner class is as follows:

Syntax: OuterClassReference.new MemberInnerClassConstructor();

#### **Internal working of Java member inner class**

The java compiler creates two class files in the case of the inner class. The class file name of the inner class is "Outer$Inner". If you want to instantiate the inner class, you must have to create the instance of the outer class. In such a case, an instance of inner class is created inside the instance of the outer class.

### Java Anonymous inner class

A normal class can implement any number of interfaces but the anonymous inner class can implement only one interface at a time.

A regular class can extend a class and implement any number of interfaces simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time. For regular/normal class, we can write any number of constructors but we can’t write any constructor for anonymous Inner class because the anonymous class does not have any name and while defining constructor class name and constructor name must be same.

**Accessing Local Variables of the Enclosing Scope, and Declaring and Accessing Members of the Anonymous Class**

Like local classes, anonymous classes can capture variables; they have the same access to local variables of the enclosing scope:

* An anonymous class has access to the members of its enclosing class.
* An anonymous class cannot access local variables in its enclosing scope that are not declared as final or effectively final.
* Like a nested class, a declaration of a type (such as a variable) in anonymous class shadows any other declarations in the enclosing scope that have the same name.
* Anonymous classes also have the same restrictions as local classes with respect to their members:
* We cannot declare static initializers or member interfaces in an anonymous class.
* An anonymous class can have static members provided that they are constant variables.

Note: We can declare the following in anonymous classes as follows:

* Fields
* Extra methods (even if they do not implement any methods of the supertype)
* Instance initializers
* Local classes

Ways:

**Anonymous inner classes are generic created via below listed two ways as follows:**

* Class (may be abstract or concrete)
* Interface

### **Java Local inner class**

Local Inner Classes are the inner classes that are defined inside a block. Generally, this block is a method body. Sometimes this block can be a for loop, or an if clause. Local Inner classes are not a member of any enclosing classes. They belong to the block they are defined within, due to which local inner classes cannot have any access modifiers associated with them. However, they can be marked as final or abstract. These classes have access to the fields of the class enclosing it.

If you want to invoke the methods of the local inner class, you must instantiate this class inside the method.

###### **Rule: Local variables can't be private, public, or protected.**

#### **Rules for Java Local Inner class**

###### **1) Local inner class cannot be invoked from outside the method.**

###### **2) Local inner class cannot access non-final local variable till JDK 1.7. Since JDK 1.8, it is possible to access the non-final local variable in the local inner class.**

### **Java static nested class**

### A static class is a class that is created inside a class, is called a static nested class in Java. It cannot access non-static data members and methods. It can be accessed by outer class name.

### It can access static data members of the outer class, including private.

### The static nested class cannot access non-static (instance) data members or

|  |  |
| --- | --- |
| Normal/Regular inner class | Static nested class |
| Without an outer class object existing, there cannot be an inner class object. That is, the inner class object is always associated with the outer class object. | Without an outer class object existing, there may be a static nested class object. That is, static nested class object is not associated with the outer class object. |
| Inside normal/regular inner class, static members can’t be declared. | Inside static nested class, static members can be declared. |
| As main() method can’t be declared, regular inner class can’t be invoked directly from the command prompt. | As main() method can be declared, the static nested class can be invoked directly from the command prompt. |
| Both static and non static members of outer class can be accessed directly. | Only a static member of outer class can be accessed directly. |

## Java Inner Interface

An interface, i.e., declared within another interface or class, is known as a nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred to by the outer interface or class. It can't be accessed directly.

**Points to remember for nested interfaces-**

* Nested interfaces declared inside class can take any access modifier, however nested interface declared inside interface is public implicitly.
* Nested interfaces are static by default, irrespective of you declare it static or not.
* Nested interfaces are accessed using outer interface or class name.
* Nested interfaces declared inside a class can have any access modifier, while nested interfaces declared inside another interfaces are public by default.
* Classes implementing inner interface are required to implement only inner interface methods, not outer interface methods.
* Classes implementing outer interface are required to implement only outer interface methods, not inner interface methods.

**Q.Does java compiler creates separate .class file for nested interfaces ?**

Yes, Java compiler creates separate .class file for nested interfaces, whose name is made with combination of outer and inner interface name with $ sign in between. For example the name of .class file in above declaration would be MyInterface$MyInnerInterface.class.

**\*Why do we use nested interface in java**

There are couple of reasons for using nested interfaces in java :

* Nesting of interfaces is a way of logically grouping interfaces which are related or used at one place only.
* Nesting of interfaces helps to write more readable and maintainable code.
* It also increases encapsulation.

**\*\*Notes-**

* An interface or class can have any number of inner interfaces inside it.
* Inner interface can extend it's outer interface.
* Inner and outer interfaces can have method and variables with same name.
* Nested interfaces can also have default and static methods from java 8 onward.
* An inner class defined inside an interface can implement the interface.
* Nesting of interfaces can be done any number of times. As a good practice you should avoid doing nesting more than once.

# 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.

1. **Let’s summarize the discussion in points:**  
   1. The main purpose of multithreading is to provide simultaneous execution of two or more parts of a program to maximum utilize the CPU time. A multithreaded program contains two or more parts that can run concurrently. Each such part of a program called thread.
2. Threads are lightweight sub-processes, they share the common memory space. In Multithreaded environment, programs that are benefited from multithreading, utilize the maximum CPU time so that the idle time can be kept to minimum.
3. A thread can be in one of the following states:  
   **NEW** – A thread that has not yet started is in this state.  
   **RUNNABLE** – A thread executing in the Java virtual machine is in this state.  
   **BLOCKED** – A thread that is blocked waiting for a monitor lock is in this state.  
   **WAITING** – A thread that is waiting indefinitely for another thread to perform a particular action is in this state.  
   **TIMED\_WAITING** – A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.  
   **TERMINATED** – A thread that has exited is in this state.  
   A thread can be in only one state at a given point in time.

### **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.**

## **Multitasking vs Multithreading vs Multiprocessing vs parallel processing**

**Multitasking:**Ability to execute more than one task at the same time is known as multitasking.

**Multithreading:**We already discussed about it. It is a process of executing multiple threads simultaneously. Multithreading is also known as Thread-based Multitasking.

**Multiprocessing:** It is same as multitasking, however in multiprocessing more than one CPUs are involved. On the other hand one CPU is involved in multitasking.

**Parallel Processing:** It refers to the utilization of multiple CPUs in a single computer system.

## **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.

 There is context-switching between the threads. There can be multiple processes inside the [OS](https://www.javatpoint.com/os-tutorial), and one process can have multiple threads.

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

### **Life Cycle of a Thread**

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**

1. **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.
2. **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**.

* **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.

1. **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.

1. **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.
2. **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.

### **Implementation of Thread States**

In Java, one can get the current state of a thread using the **Thread.getState()** method. The **java.lang.Thread.State** class of Java provides the constants ENUM to represent the state of a thread. These constants are:

1. **public** **static** **final** Thread.State NEW

It represents the first state of a thread that is the NEW state.

1. **public** **static** **final** Thread.State RUNNABLE

It represents the runnable state.It means a thread is waiting in the queue to run.

1. **public** **static** **final** Thread.State BLOCKED

It represents the blocked state. In this state, the thread is waiting to acquire a lock.

1. **public** **static** **final** Thread.State WAITING

It represents the waiting state. A thread will go to this state when it invokes the Object.wait() method, or Thread.join() method with no timeout. A thread in the waiting state is waiting for another thread to complete its task.

1. **public** **static** **final** Thread.State TIMED\_WAITING

It represents the timed waiting state. The main difference between waiting and timed waiting is the time constraint. Waiting has no time constraint, whereas timed waiting has the time constraint. A thread invoking the following method reaches the timed waiting state.

* sleep
* join with timeout
* wait with timeout
* parkUntil
* parkNanos

1. **public** **static** **final** Thread.State TERMINATED

It represents the final state of a thread that is terminated or dead. A terminated thread means it has completed its execution.

### **Thread Priorities**

* Thread priorities are the integers which decide how one thread should be treated with respect to the others.
* Thread priority decides when to switch from one running thread to another, process is called context switching
* A thread can voluntarily release control and the highest priority thread that is ready to run is given the CPU.
* A thread can be preempted by a higher priority thread no matter what the lower priority thread is doing. Whenever a higher priority thread wants to run it does.
* To set the priority of the thread setPriority() method is used which is a method of the class Thread Class.
* In place of defining the priority in integers, we can use MIN\_PRIORITY, NORM\_PRIORITY or MAX\_PRIORITY.

#### **Priority of a Thread (Thread Priority)**

Each thread has a priority. Priorities are represented by a number between 1 and 10. In most cases, the thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses. Note that not only JVM a Java programmer can also assign the priorities of a thread explicitly in a Java program.

##### Setter & Getter Method of Thread Priority

Let's discuss the setter and getter method of the thread priority.

**public final int getPriority():** The java.lang.Thread.getPriority() method returns the priority of the given thread.

**public final void setPriority(int newPriority):** The java.lang.Thread.setPriority() method updates or assign the priority of the thread to newPriority. The method throws IllegalArgumentException if the value newPriority goes out of the range, which is 1 (minimum) to 10 (maximum).

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

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

Default priority of a thread is 5 (NORM\_PRIORITY). The value of MIN\_PRIORITY is 1 and the value of MAX\_PRIORITY is 10.

###### **Methods: isAlive() and join()**

* In all the practical situations main thread should finish last else other threads which have spawned from the main thread will also finish.
* To know whether the thread has finished we can call isAlive() on the thread which returns true if the thread is not finished.
* Another way to achieve this by using join() method, this method when called from the parent thread makes parent thread wait till child thread terminates.
* These methods are defined in the Thread class.
* We have used isAlive() method in the above examples too.

### **Synchronization**

* Multithreading introduces asynchronous behavior to the programs. If a thread is writing some data another thread may be reading the same data at that time. This may bring inconsistency.
* When two or more threads need access to a shared resource there should be some way that the resource will be used only by one resource at a time. The process to achieve this is called synchronization.
* To implement the synchronous behavior java has synchronous method. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. All the other threads then wait until the first thread come out of the synchronized block.
* When we want to synchronize access to objects of a class which was not designed for the multithreaded access and the code of the method which needs to be accessed synchronously is not available with us, in this case we cannot add the synchronized to the appropriate methods. In java we have the solution for this, put the calls to the methods (which needs to be synchronized) defined by this class inside a synchronized block in following manner.

### **Inter-thread Communication**

We have few methods through which java threads can communicate with each other. These methods are wait(), notify(), notifyAll(). All these methods can only be called from within a synchronized method.

1. To understand synchronization java has a concept of monitor. Monitor can be thought of as a box which can hold only one thread. Once a thread enters the monitor all the other threads have to wait until that thread exits the monitor.
2. wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().
3. notify() wakes up the first thread that called wait() on the same object. notifyAll() wakes up all the threads that called wait() on the same object. The highest priority thread will run first.

### **Creating a thread in Java**

There are two ways to create a thread in Java:

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

**Thread Class vs Runnable Interface**

1. If we extend the Thread class, our class cannot extend any other class because Java doesn’t support multiple inheritance. But, if we implement the Runnable interface, our class can still extend other base classes.
2. We can achieve basic functionality of a thread by extending Thread class because it provides some inbuilt methods like yield(), interrupt() etc. that are not available in Runnable interface.
3. Using runnable will give you an object that can be shared amongst multiple threads.

#### **Thread class**

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

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

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

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

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

#### **Runnable interface:**

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

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

#### **Starting a thread:**

The **start() method** of Thread class is used to start a newly created thread. It performs the following tasks:

* A new thread starts(with new callstack).
* The thread moves from New state to the Runnable state.
* When the thread gets a chance to execute, its target run() method will run.

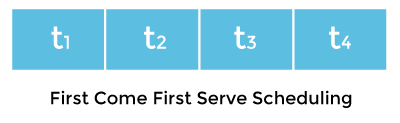
### **Thread Scheduler in Java**

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.

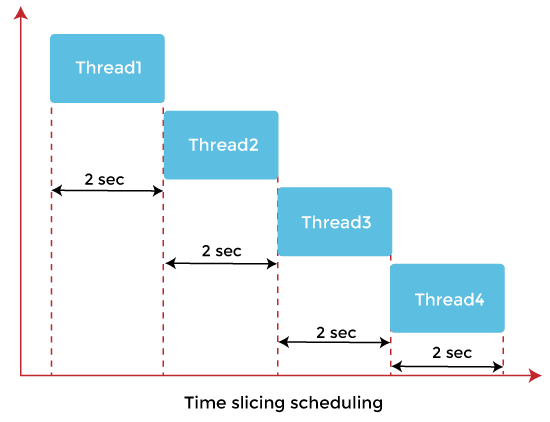
**Time of Arrival:** Suppose two threads of the same priority enter the runnable state, then priority cannot be the factor to pick a thread from these two threads. In such a case, **arrival time** of thread is considered by the thread scheduler. A thread that arrived first gets the preference over the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

#### **Thread Scheduler Algorithms**

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

##### **First Come First Serve Scheduling:**

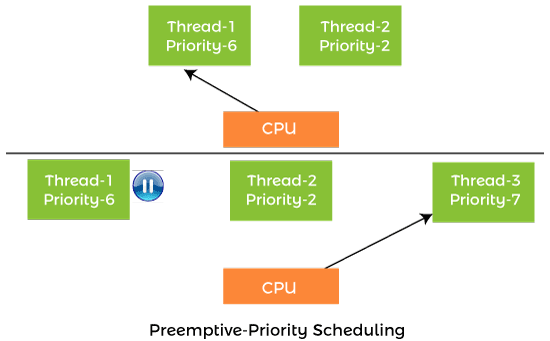
In this scheduling algorithm, the scheduler picks the threads thar arrive first in the runnable queue. Observe the following table:

In the above table, we can see that Thread t1 has arrived first, then Thread t2, then t3, and at last t4, and the order in which the threads will be processed is according to the time of arrival of threads.

Hence, Thread t1 will be processed first, and Thread t4 will be processed last.

##### **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.

In the above diagram, each thread is given a time slice of 2 seconds. Thus, after 2 seconds, the first thread leaves the CPU, and the CPU is then captured by Thread2. The same process repeats for the other threads too.

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

##### The name of the scheduling algorithm denotes that the algorithm is related to the priority of the threads.

Suppose there are multiple threads available in the runnable state. The thread scheduler picks that thread that has the highest priority. Since the algorithm is also preemptive, therefore, time slices are also provided to the threads to avoid starvation. Thus, after some time, even if the highest priority thread has not completed its job, it has to release the CPU because of preemption.

#### **Working of the Java Thread Scheduler**Thread Scheduler in Java

Let's understand the working of the Java thread scheduler. Suppose, there are five threads that have different arrival times and different priorities. Now, it is the responsibility of the thread scheduler to decide which thread will get the CPU first.

The thread scheduler selects the thread that has the highest priority, and the thread begins the execution of the job. If a thread is already in runnable state and another thread (that has higher priority) reaches in the runnable state, then the current thread is pre-empted from the processor, and the arrived thread with higher priority gets the CPU time.

When two threads (Thread 2 and Thread 3) having the same priorities and arrival time, the scheduling will be decided on the basis of FCFS algorithm. Thus, the thread that arrives first gets the opportunity to execute first.

### **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.

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

##### **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.

### **Main thread in Java**

Java provides built-in support for multithreaded programming. A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution.

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program because it is the one that is executed when our program begins.

There are certain properties associated with the main thread which are as follows:

• It is the thread from which other “child” threads will be spawned.

• Often, it must be the last thread to finish execution because it performs various shutdown actions

### **Daemon Thread in Java**

Daemon thread in Java is a low-priority thread that runs in the background to perform tasks such as garbage collection. Daemon thread in Java is also a service provider thread that provides services to the user thread. Its life depends on the mercy of user threads i.e. when all the user threads die, JVM terminates this thread automatically.

In simple words, we can say that it provides services to user threads for background supporting tasks. It has no role in life other than to serve user threads.

**Example of Daemon Thread in Java:**Garbage collection in Java (gc), finalizer, etc.

**Properties of Java Daemon Thread**

* They cannot prevent the JVM from exiting when all the user threads finish their execution.
* JVM terminates itself when all user threads finish their execution.
* If JVM finds a running daemon thread, it terminates the thread and, after that, shutdown it. JVM does not care whether the Daemon thread is running or not.
* It is an utmost low priority thread.

This clearly shows that we cannot call the setDaemon() method after starting the thread.

#### **Daemon vs. User Threads**

1. **Priority:** When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
2. **Usage:** Daemon thread is to provide services to user thread for background supporting task.

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

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

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

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

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

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

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

### **Thread Group in Java**

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

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

Java thread group is implemented by *java.lang.ThreadGroup* class.

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

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

#### Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

|  |  |  |
| --- | --- | --- |
| No. | Constructor | Description |
| 1) | ThreadGroup(String name) | creates a thread group with given name. |
| 2) | ThreadGroup(ThreadGroup parent, String name) | creates a thread group with a given parent group and name. |

#### Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods is given below.

|  |  |  |  |
| --- | --- | --- | --- |
| S.N. | Modifier and Type | Method | Description |
| 1) | void | [checkAccess()](https://www.javatpoint.com/java-threadgroup-checkaccess-method) | This method determines if the currently running thread has permission to modify the thread group. |
| 2) | int | [activeCount()](https://www.javatpoint.com/java-threadgroup-activecount-method) | This method returns an estimate of the number of active threads in the thread group and its subgroups. |
| 3) | int | [activeGroupCount()](https://www.javatpoint.com/java-threadgroup-activegroupcount-method) | This method returns an estimate of the number of active groups in the thread group and its subgroups. |
| 4) | void | [destroy()](https://www.javatpoint.com/java-threadgroup-destroy-method) | This method destroys the thread group and all of its subgroups. |
| 5) | int | [enumerate(Thread[] list)](https://www.javatpoint.com/java-threadgroup-enumerate-method) | This method copies into the specified array every active thread in the thread group and its subgroups. |
| 6) | int | [getMaxPriority()](https://www.javatpoint.com/java-threadgroup-getmaxpriority-method) | This method returns the maximum priority of the thread group. |
| 7) | String | [getName()](https://www.javatpoint.com/java-threadgroup-getname-method) | This method returns the name of the thread group. |
| 8) | ThreadGroup | [getParent()](https://www.javatpoint.com/java-threadgroup-getparent-method) | This method returns the parent of the thread group. |
| 9) | void | [interrupt()](https://www.javatpoint.com/java-threadgroup-interrupt-method) | This method interrupts all threads in the thread group. |
| 10) | boolean | [isDaemon()](https://www.javatpoint.com/java-threadgroup-isdaemon-method) | This method tests if the thread group is a daemon thread group. |
| 11) | void | [setDaemon(boolean daemon)](https://www.javatpoint.com/java-threadgroup-setdaemon-method) | This method changes the daemon status of the thread group. |
| 12) | boolean | [isDestroyed()](https://www.javatpoint.com/java-threadgroup-isdestroyed-method) | This method tests if this thread group has been destroyed. |
| 13) | void | [list()](https://www.javatpoint.com/java-threadgroup-list-method) | This method prints information about the thread group to the standard output. |
| 14) | boolean | [parentOf(ThreadGroup g](https://www.javatpoint.com/java-threadgroup-parentof-method) | This method tests if the thread group is either the thread group argument or one of its ancestor thread groups. |
| 15) | void | [suspend()](https://www.javatpoint.com/java-threadgroup-suspend-method) | This method is used to suspend all threads in the thread group. |
| 16) | void | [resume()](https://www.javatpoint.com/java-threadgroup-resume-method) | This method is used to resume all threads in the thread group which was suspended using suspend() method. |
| 17) | void | [setMaxPriority(int pri)](https://www.javatpoint.com/java-threadgroup-setmaxpriority-method) | This method sets the maximum priority of the group. |
| 18) | void | [stop()](https://www.javatpoint.com/java-threadgroup-stop-method) | This method is used to stop all threads in the thread group. |
| 19) | String | [toString()](https://www.javatpoint.com/java-threadgroup-tostring-method) | This method returns a string representation of the Thread group. |

# Java Garbage Collection

In java, garbage means unreferenced objects.

Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

### **Advantage of Garbage Collection**

* It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
* It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

## **How can an object be unreferenced?**

There are many ways:

* By nulling the reference
* By assigning a reference to another
* By anonymous object etc.

### **1) By nulling a reference:**

Employee e=**new** Employee();

e=**null**;

### **2) By assigning a reference to another:**

Employee e1=**new** Employee();

Employee e2=**new** Employee();

e1=e2;//now the first object referred by e1 is available for garbage collection

### **3) By anonymous object:**

1. **new** Employee();

## **finalize() method**

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

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

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

## **gc() method**

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

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

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

# Synchronization in Java

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

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

### **Why use Synchronization?**

The synchronization is mainly used to

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

### **Types of Synchronization**

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Here, we will discuss only thread synchronization.

### **Thread Synchronization**

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

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

### **Mutual Exclusive**

Mutual Exclusive helps keep threads from interfering with one another while sharing data. It can be achieved by using the following three ways:

1. By Using Synchronized Method
2. By Using Synchronized Block
3. By Using Static Synchronization

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

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

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

### **Java Synchronized Method**

If you declare any method as synchronized, it is known as synchronized method. Synchronized method is used to lock an object for any shared resource.

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

# Synchronized Block in Java

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

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

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

### **Points to Remember**

* Synchronized block is used to lock an object for any shared resource.
* Scope of synchronized block is smaller than the method.
* A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
* The system performance may degrade because of the slower working of synchronized keyword.
* Java synchronized block is more efficient than Java synchronized method.

### **Syntax**

**synchronized** (object reference expression) {

  //code block

}