Java Basics:

**The Basic Syntax:**

**Comments in Java**

There are three types of comments in Java.

**i.**Single line Comment

// System.out.println("");

**ii.** Multi-line Comment

/\*

System.out.println("");

System.out.println("");

\*/

**iii.**Documentation Comment. Also called a **doc comment**.

/\*\* documentation \*/

**main() method:**

In the Java programming language, every application must contain a main method whose signature is:

public static void main(String[] args)

* **public**: So that [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/) can execute the method from anywhere.

Note : JVM execute the main method

* **static**: The main method is to be called without an object. The modifiers public and static can be written in either order.
* **void**: The main method doesn’t return anything.
* **main()**: Name configured in the JVM.
* **String[]**: The main method accepts a single argument, i.e., an array of elements of type String.

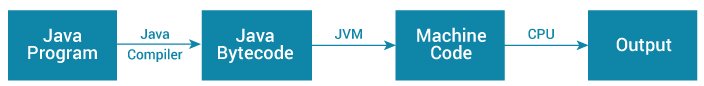
In Java, all codes must reside inside a class, and there is at most one public class which contains the main() method.

**Q. Can we print something on the console without creating main() method?**

It is very important question from the interview’s perceptive point.

The answer is yes, we can print if we are using JDK version 1.6 or previous and if after that that it will throw an error.

**Java JDK, JRE and JVM:**



.java .class [interpreter](https://www.geeksforgeeks.org/compiler-vs-interpreter-2/)

{Platform independent}

JVM is present at different OS (machine) and is an abstract machine that enables your computer to run a Java program.



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

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

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

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

**How JVM Works – JVM Architecture?**

**JVM** (Java Virtual Machine / interpreter) acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of JRE (Java Runtime Environment).

Java applications are called WORA (Write Once Run Anywhere). This means a programmer can develop Java code on one system and can expect it to run on any other Java-enabled system without any adjustment. This is all possible because of JVM.

When we compile a *.java* file, *.class* files (contains byte-code) with the same class names present in *.java* file are generated by the Java compiler. This *.class* file goes into various steps when we run it. These steps together describe the whole JVM.

------------------------------------------------------------------------------------------OR

JDK vs. JRE vs. JVM: Key differences:

And now, for the differences:

* JDK, or Java Development Kit, is a software development kit. It is platform-specific, so separate installers are needed for each operating system (e.g., Mac, Unix and Windows).

JDK is the development platform, while JRE is for execution.

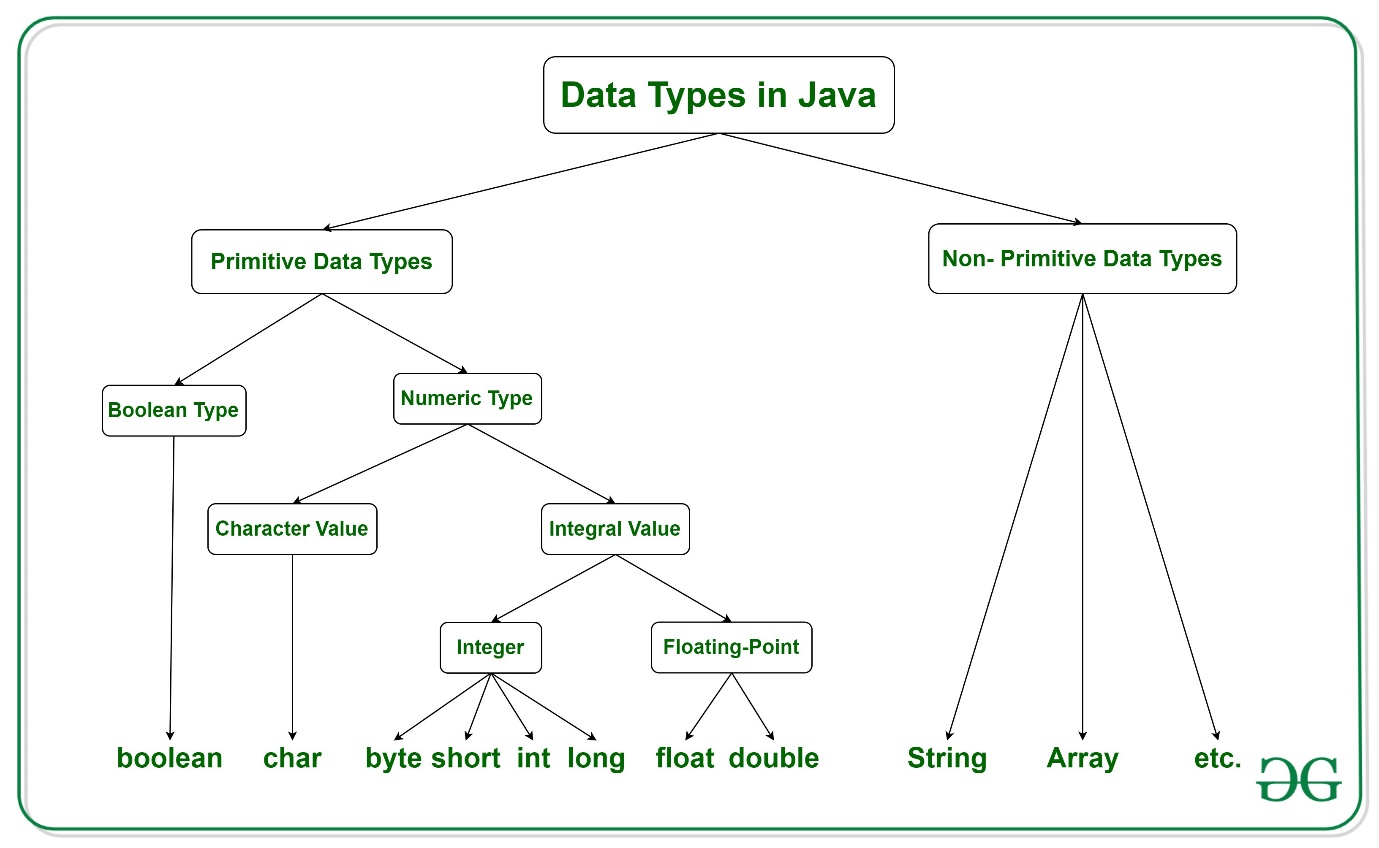
* JVM is the foundation, or the heart of Java programming language, and ensures the program’s Java source code will be platform-agnostic. Java Virtual Machine, or JVM, loads, verifies and executes Java bytecode. It is known as the interpreter or the core of Java programming language because it executes Java programming.
* JRE, or [Java Runtime Environment](https://www.ibm.com/cloud/learn/jre), is a set of software tools responsible for execution of the Java program or application on your system. JRE uses heap space for dynamic memory allocation for Java objects. JRE is also used in JDB (Java Debugging).

--------------------------------------------------------------------------------------------------------------------------------------

Complementary technologies

There are many complementary technologies that can be used to enhance JVM, JRE or JDK. The following technologies are among the most frequently used:

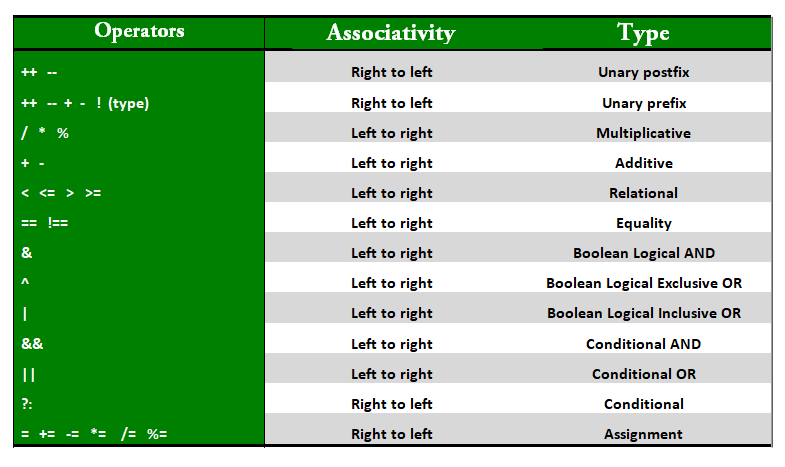
* **Just-in-time Compiler (JIT)** is part of JVM and optimizes the conversion of bytecode to machine code. It selects similar bytecodes to compile at the same time, reducing the overall duration of bytecode to machine code compilation.
* **Javac**, another complementary tool, is a compiler that reads Java definitions and translates them into bytecode that can run on JVM.
* **Javadoc** converts API documentation from Java source code to HTML. This is useful when creating standard documentation in HTML.





**char**: The char data type is a single 16-bit Unicode character.

**Why is the size of char 2 bytes in java?**  
In other languages like C/C++ uses only ASCII characters and to represent all ASCII characters 8-bits is enough,   
But java uses the Unicode system not the ASCII code system and to represent Unicode system 8 bit is not enough to represent all characters so java uses 2 bytes for characters.  
***Unicode****defines a fully international character set that can represent most of the world’s written languages. It is a unification of dozens of character sets, such as Latin, Greeks,* *Cyrillic, Katakana, Arabic*,*and many more.*



**Jump Statements in Java - break, continue and ‘return’**

**Return** statement not required (but can be used) for **methods** with return type void. We can use “return;” which means not return anything.

# Object Oriented Programming (OOPs) Concept in Java

OOPs Concepts are as follows:

1. Class
2. Object
3. Method and method passing
4. Pillars of OOPS
   * Abstraction
   * Encapsulation
   * Inheritance
   * Polymorphism
     + Compile-time polymorphism
     + Run-time polymorphism



### **Class**

### A class is a user defined blueprint or prototype from which objects are created.  It represents the set of properties or methods that are common to all objects of one type.

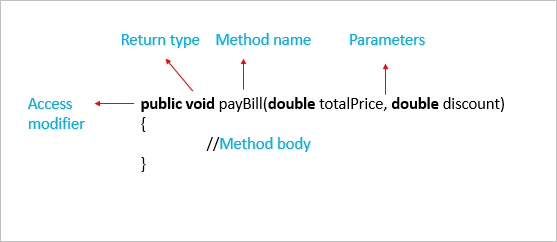
### **Object**

It is a basic unit of Object-Oriented Programming and represents the real-life entities.  A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of:

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

**Method in java:**





**Access Modifiers:**

Modifiers control the scope of a class, constructor, variable, method, or data member.

**Access Modifiers** are divided into*two* groups:

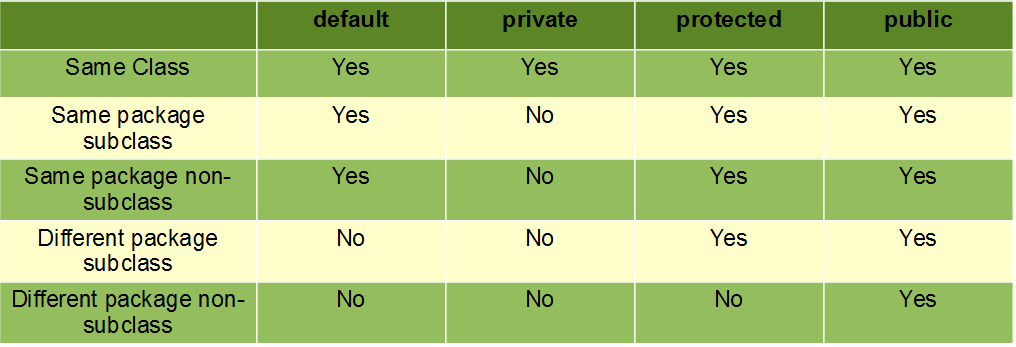
1. **Access Modifiers:** controls the access level. E.g., default, public, protected, private
2. **Non-access Modifiers:** do not control access level but ‘provide other functionality’. E.g., static, final, abstract

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.

**OR**

* **Public**: Accessible from any other class
* **Private**: Accessible only inside its own class
* **Protected**: Accessible inside the same package and to the sub-classes in different packages.
* **Default**: Accessible inside the same package. Members created without any access modifier will have this access.



**There are some Java Non-Access Modifiers:**

1. **static:**The member belongs to the class, not to objects of that class.
2. **final:** Variable values can’t be changed once assigned, methods can’t be overridden, classes can’t be inherited.
3. **abstract:** If applied to a method – has to be implemented in a subclass, if applied to a class – contains abstract methods
4. **synchronized:** Controls thread access to a block/method.
5. **volatile:**The variable value is always read from the main memory, not from a specific thread’s memory.
6. **transient:** The member is skipped when serializing an object.

For **classes**, you can use either final or abstract:

|  |  |  |
| --- | --- | --- |
| **Modifier** | **Description** |  |
| final | The class cannot be inherited by other classes. |  |
| abstract | The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.) |  |

For **attributes and methods**, you can use the one of the following:

|  |  |
| --- | --- |
| **Modifier** | **Description** |
| final | Attributes and methods cannot be overridden/modified |
| static | Attributes and methods belong to the class, rather than an object |
| abstract | Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from). |

**Constructors in Java:**

Constructors are used to initializing the object’s state. Constructors are used to assigning values to the class variables (or instance variables) at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

**When is a Constructor called?**

Each time an object is created using a **new()** keyword, at least one constructor (it could be the default constructor) is invoked to assign initial values to the **data members**of the same class. Primarily there are two types of constructors in java:

* **No-argument/ non-parameterized constructor:**

A constructor that has no parameter is known as the default constructor. If we don’t define a constructor in a class, then the compiler creates **default constructor (with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.

* **Parameterized Constructor:**

A constructor that has parameters

**How constructors are different from methods in Java?**

* Constructors must have the **same name** as the class within which it is defined while it is not necessary for the method in java.
* Constructors do not return any type (**No return type**) while method(s) have the return type or **void** if does not return any value.
* Constructors are **called only once** at the time of Object creation while method(s) can be called any number of times.

**Constructor Chaining in Java:**

Constructor chaining is the process of calling one constructor from another constructor with respect to current object.   
Constructor chaining can be done in two ways: 

* **Within same class**: It can be done using **this()** keyword for constructors in same class
* **From base class:**by using **super()** keyword to call constructor from the base/parent class.

**‘this’ reference in Java:**

‘this’ is a reference variable that refers to the current object.

Following are the ways to use ‘this’ keyword in java:

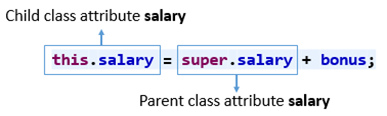
1. Using ‘**this’** keyword to refer current class instance variables
2. Using this() to invoke/call current class constructor
3. Using ‘**this’** keyword to return the current class instance
4. **Using ‘this’ keyword as method parameter**
5. **Using ‘this’ keyword to invoke current class method**
6. **Using ‘this’ keyword as an argument in the constructor call**

**Super:**

Super keyword is used to invoke/call the constructors of parent/ base/ super class.

super keyword can also be used for

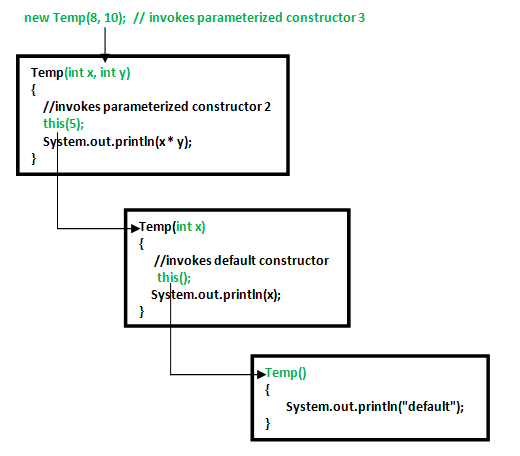
* invoking a parent class method from a child class method
* accessing a parent class instance variable in the child class in case there is a variable in the child class also with the same name

****

Constructor chaining occurs through **inheritance**. A sub class constructor’s task is to call super class’s constructor first. This ensures that creation of sub class’s object starts with the initialization of the data members of the super class. There could be any numbers of classes in inheritance chain. Every constructor calls up the chain till class at the top is reached.

**Why do we need constructor chaining ?**   
This process is used when we want to perform multiple tasks in a single constructor rather than creating a code for each task in a single constructor we create a separate constructor for each task and make their chain which makes the program more readable. 

1. **Constructor Chaining within same class using this() keyword : -**



**Rules of constructor chaining :**

1. The **this()** expression should always be the first line of the constructor.
2. There should be at-least be one constructor without the this() keyword (constructor 3 in above example).
3. Constructor chaining can be achieved in any order.

**What happens if we change the order of constructors?**  
Nothing, Constructor chaining can be achieved in any order

1. **Constructor Chaining to other class using super() keyword : -**

Note: Similar to constructor chaining in same class, **super()** should be the first line of the constructor as super class’s constructor are invoked before the sub class’s constructor.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**Alternative method - using Init block**:   
When we want certain common resources to be executed with every constructor we can put the code in the **init block**. Init block is always executed before any constructor, whenever a constructor is used for creating a new object.

# **The Initializer Block in Java:**

Initializer block contains the code that is always executed whenever an instance is created. It is used to declare/initialize the common part of various constructors of a class. The order of initialization constructors and initializer block doesn’t matter, initializer block is always executed before constructor.

**What if we want to execute some code once for all objects of a class?**   
We use Static Block in Java

**Static blocks:**

**Static blocks** are used to initialize static variables when it cannot be done in a single line. They can also be used to add pre-processing if required. If you need to do the computation in order to initialize your **static variables**, you can declare a static block - that gets executed exactly once, when the class is first loaded.

Java supports a special block, called a static block (also called static clause) that can be used for static initialization of a class. This code inside the static block is executed only once: the first time the class is loaded into memory.

**static Keyword in Java:**

The static keyword in Java is mainly used for memory management. The static keyword in Java is used to share the same variable or method of a given class. The static keyword belongs to the class than an instance of the class. The static keyword is used for a constant variable or a method that is the same for every instance of a class.

***Note:****To create a static member (block, variable, method, nested class), precede its declaration with the keyword static.*

**The *static* keyword is a non-access modifier in Java that is applicable for the following:**

1. **Blocks** **-** **Static blocks** are used to initialize static variables when it cannot be done in a single line. They can also be used to add pre-processing if required. If you need to do the computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded.
2. **Variables -** When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are, essentially, *global variables*. (All instances of the class share the same static variable.)

**Important points for static variables:**

* We can create static variables at the class level only.
* *static block* and *static variables* are executed in the order they are present in a program.

1. **Methods -** When a method is declared with the *static* keyword, it is known as the static method. The most common example of a static method is the *main( )* method. As discussed above; Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

* They can only directly call other static methods.
* They can only directly access static data.
* They cannot refer to **this** or **super** in any way.

1. **Classes** **-** A class can be made **static** only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. *Nested static class* doesn’t need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

**When to use static variables and methods?**

Use the static variable for the property that is common to all objects. For example, in class Student, all students share the same college name. Use static methods for changing static variables.

**Memory Management:**

Allocation and deallocation of memory is a critical task and requires a lot of care and consideration.

However, in Java, unlike other programming language, the JVM and to be specific Garbage Collector has the role of managing memory allocation so that the programmer needs not to.

Whereas in other programming languages such as C the programmer has direct access to the memory who allocates memory in his code, thereby creating a lot of scope for leaks.

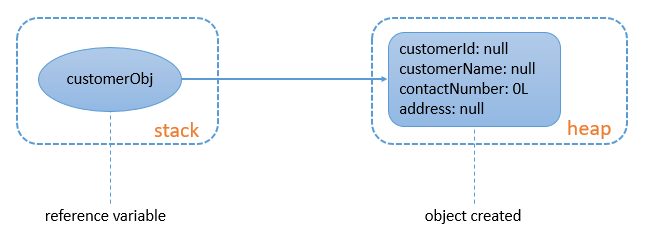


**Garbage Collector Overview**

When a program executes in Java, it uses memory in different ways. The heap is a part of memory where objects live. It’s the only part of memory that involved in the garbage collection process. It is also known as garbage collectible heap. All the garbage collection makes sure that the heap has as much free space as possible. The function of the garbage collector is to find and delete the objects that cannot be reached.

The memory is logically divided into two primary sections - Stack and Heap.

* All local variables (reference variables are also local variables. Reference variables are local variables which stores the address of another memory location.) and method invocations are stored in the **stack**
* All objects along with their instance variables are stored in the **heap**

****

Notes:

* One reference variable can point to one and only one object at a time.
* One object can be referenced by multiple reference variables at any given point of time.

Sometimes, even though a resource in a program is unreachable or not in use, the memory used by that resource is not deallocated. This is called **Memory leak** and is undesirable.

In some languages, it is the programmer's responsibility for deallocating the memory occupied by such resources.

Java, on the other hand, has a garbage collector which automatically deallocates the memory used by such resources. This prevents memory leak.

When an object does not have any reference, it becomes eligible for garbage collection.

Let us look at some of the possibilities.

**Objects eligible for garbage collection:**

* + When the reference variable pointing to the object is initialized to null, the object will not have any reference.
  + When the reference variable is initialized to a new object and there is no reference to the previous object.
  + When a reference variable is local to some method, it will be removed from the stack as soon as the method finishes execution. The object pointed by the reference variable then becomes eligible for garbage collection.

**Encapsulation:**

The meaning of **Encapsulation**, is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

* + declare class variables/attributes as private
  + provide public **get** and **set** methods to access and update the value of a private variable

**Getter and Setter in Java:**

The get method returns the variable value, and the set method sets the value.

**Some of the most important benefits of using getters and setters:**

* **It helps us achieve encapsulation which is used to hide the state of a structured data object inside a class, preventing unauthorized direct access to them**
* Achieve immutability by declaring the fields as private and using only getters
* **Getters and setters also allow additional functionalities like validation, error handling that could be added more easily in the future. Thus, we can add conditional logic and provide behaviour according to the needs**
* We can provide different access levels to the fields; for example, the get (read-access) may be public, while the set (write-access) could be protected
* Control over setting the value of the property correctly
* With getters and setters, we achieve one more key principle of OOP, i.e., abstraction, which is hiding implementation details so that no one can use the fields directly in other classes or modules

# **Inheritance in Java**

Java Inheritance (Subclass and Superclass):

In Java, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

* **subclass** (child) - the class that inherits from another class
* **superclass** (parent) - the class being inherited from

To inherit from a class, use the extends keyword.

For e.g., the Car class (subclass/ child class) inherits the attributes and methods from the Vehicle class (superclass/ parent class)

\*Note: If you don't want other classes to inherit from a class, use the final keyword.

**Constructor call in Inheritance:**

 When a child class object is created, the child class constructor invokes the parent class constructor before executing any statement present in the child constructor. i.e.

* + the parent class parameterless constructor is implicitly invoked by the child class constructors
  + In fact, the parameterless constructor of the parent class gets implicitly called by the child class constructors due to an implicit **super()** statement. This can also be done explicitly as shown below.
  + super(); *// Invoking the parent class constructor*

# Next, how are parameterized constructors of parent class invoked?

* + This can be done using **super(arguments)**
  + super(custId, custName); *// Invoking the parent class parameterized constructor*
  + note that the call to a super constructor has to be the first statement inside a constructor.

# Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance

# **Java and Multiple Inheritance**

***Note:****Java doesn’t support Multiple Inheritance*

* Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with the same signature in both the superclasses and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority.

# Polymorphism in Java

Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; **Inheritance** lets us inherit attributes and methods from another class. **Polymorphism** uses those methods to perform different tasks. This allows us to perform a single action in different ways.

Polymorphism is the ability of an object to take different forms, i.e., a single action that can be performed in different ways. So, polymorphism means many forms.

**Types of polymorphism:**

In Java polymorphism is mainly divided into two types:

* Compile-time Polymorphism / Staticpolymorphism
* Runtime Polymorphism / Dynamic polymorphism / Dynamic Method Dispatch

1. **Compile-time/** static **polymorphism:**

It is also known as static polymorphism. This type of polymorphism is achieved by method 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.

Signature can differ by

* the number of parameters
* the data type of parameters
* the order of the parameters

**Note**: We cannot overload methods by their return type, i.e., two or more methods are not overloaded if they differ only in their return type.

**Constructor Overloading:**

Just like normal methods, constructors can also be overloaded, i.e., a class can have multiple constructors. This is called **constructor overloading**.

1. **Runtime/** dynamic **polymorphism/** Dynamic Method Dispatch:

Similar to static polymorphism or compile time polymorphism, polymorphism can also be achieved at runtime. Such type of polymorphism is known as **dynamic polymorphism**. This type of polymorphism is achieved using overriding the parent method in the child class, called as **Method Overriding**. 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.

**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**.

* When we override a method in the child class, it should have the same signature as that of the parent class.
* The method should not have a weaker access modifier.
* Private methods are not overridden.

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

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.

**\* Dynamic Method Dispatch:**

Phone obj = new Phone(); // Allowed

Smartphone obj = new Smartphone(); // Allowed

Phone obj = new Smartphone(); // Allowed ultra max pro 😉

**Phone Methods: -**

**method1() => can call this**

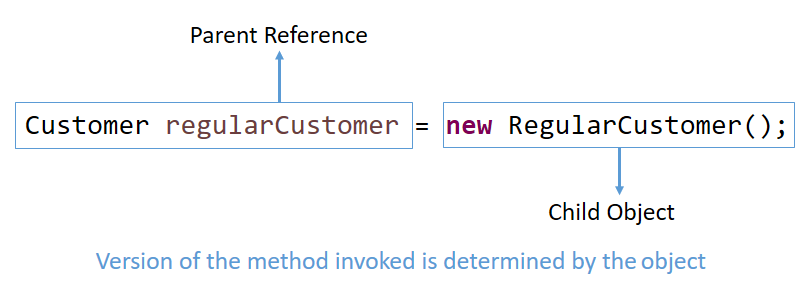
**method2() => NO**

**Smartphone Methods: -**

**method2() => this will be called //@overriden**

**method3() => cannot call this**

A parent class reference can refer to a child class object.

****

Only the overridden methods can be called using the parent class reference. Any new method created in the child class will not be accessible using the parent class reference.

The version of the method that will be called is determined by the object, and this decision is taken at runtime. This is called **Dynamic binding**.

Static methods are not overridden. They will be called based on the type of reference used.

Note: When a base class is specialized into child classes, instantiating the base class is not a good practice.

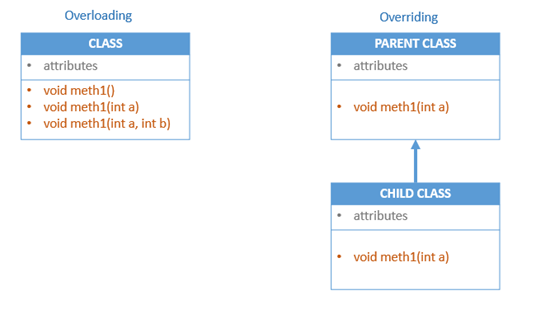
**Generic Method:**

Dynamic binding also allows the programmer to have generic methods.

Generic methods will have the parent type as the formal argument. Such methods accepting the parent type allows working with the objects of it as well as child type.

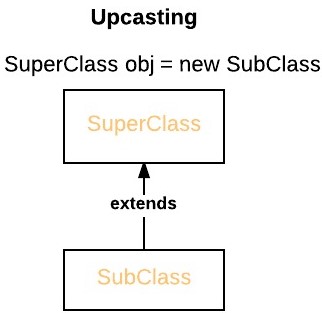
**Comparison of Method Overloading and Method Overriding:**

* Method overloading is a feature through which a class can have multiple methods with the same name but different signature.
* Method overriding is a feature that allows a subclass or child class to have a method with the same name and signature as that of the parent class.
* Overloading is an example of compile-time polymorphism as the decision about which method has to be invoked is taken during compilation time.
* Overriding is an example of runtime polymorphism as the decision about which method has to be invoked is taken during runtime.

****

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\*\*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 methods at run time.



();

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Abstraction in Java:**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.

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

**Abstract class and methods:**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.  
Abstraction can be achieved with either **abstract classes** or **interfaces**.

The abstract keyword is a non-access modifier, used for **classes** and **methods**:

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

\*An abstract class can have both abstract and regular methods.

An **abstract method** is a method without any definition, i.e., the body. The signature of an abstract method must be preceded by the abstract keyword.

public abstract double payBill(double totalPrice);

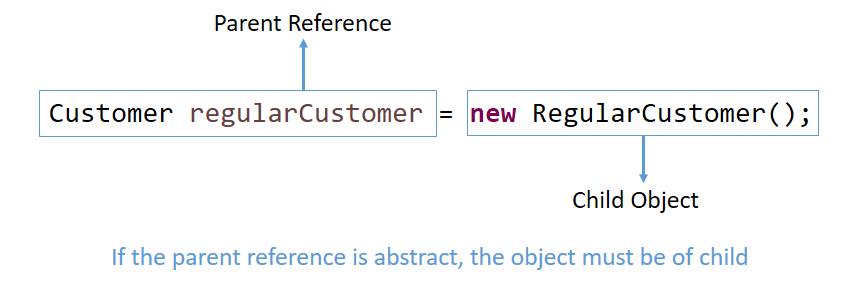
An **abstract class** is a class which is incomplete and **cannot be instantiated**.

abstract public class Customer {

}

Some points that you should be knowing about **abstract class** are:

* If a class contains at least one abstract method, the class should be abstract. i.e., If a class include abstract methods, then it must be declared abstract.
* An abstract class encapsulates the common behaviours of all its child classes with the help of abstract methods
* Concrete (non-abstract) classes which extend an abstract class must implement all the abstract methods. Otherwise, they should be made abstract as well.
* A class can be made abstract even without any abstract methods.



Thus, **abstract classes** enforce **inheritance** (since they can’t be instantiated)

and

**abstract methods** enforce **overriding** (since they are incomplete with no implementation/ body)

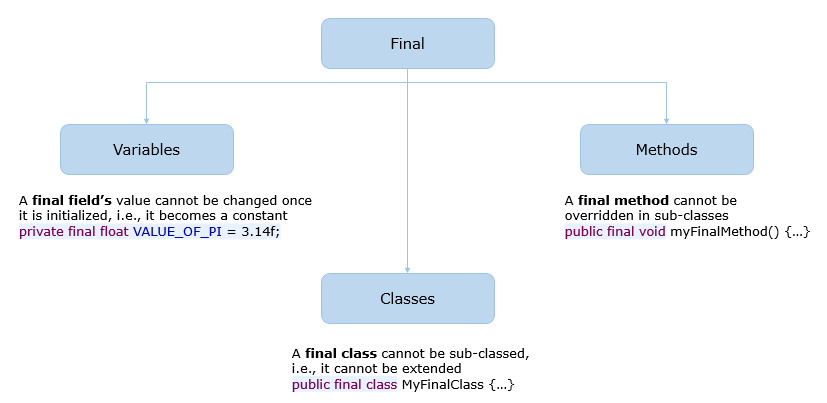
**Why can’t we create the object of an abstract class?**

Because these classes are incomplete, they have abstract methods that have no body so if java allows you to create object of this class, then if someone calls the abstract method using that object then What would happen? There would be no actual implementation of the method to invoke.  
Also because an object is concrete. An abstract class is like a template, so you have to extend it and build on it before you can use it.

**final keyword:**

In programming, we may encounter situations where we may have to create components that must remain constant, i.e., never change. In such cases, the final keyword can be used.

The final keyword can be used with classes, variables and methods. The details are diagrammatically shown below.



## Interfaces in Java:

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

interface <interface\_name> {

// declare constant fields

// declare methods that abstract by default.

}

To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in an interface are declared with an empty body and are **public** and all fields are **public**, **static** and **final** by default. A class that implements an interface *must implement all the methods* declared in the interface. To implement interface use **implements** keyword.

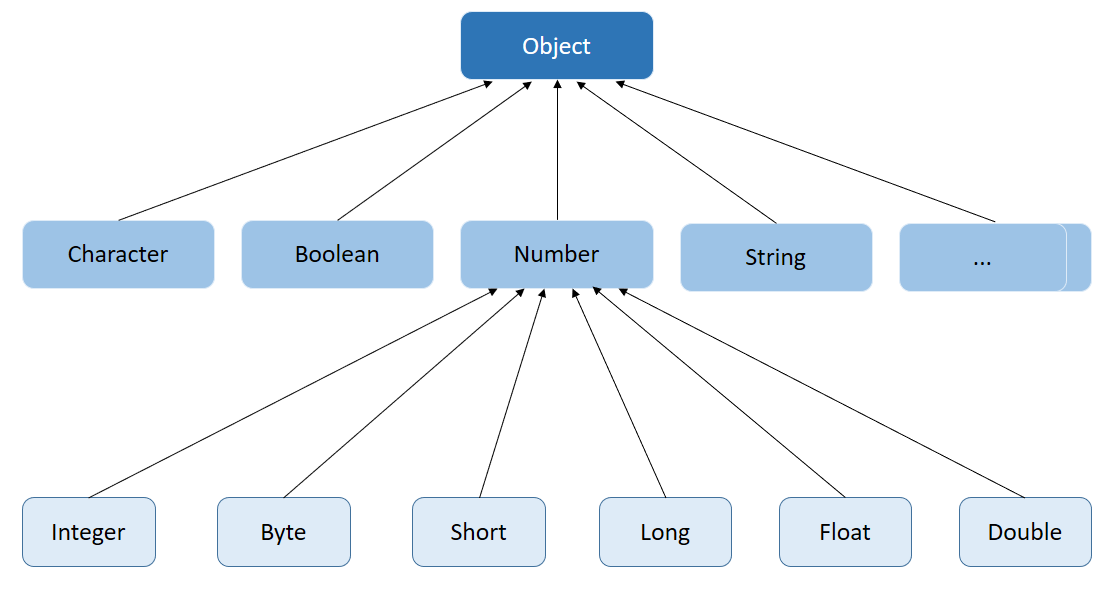
\*\*Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance.

An interface is used to define a generic template which can be implemented by various classes.

* It contains method signatures and constant declarations
* The methods declared in an interface are implicitly public and abstract and the data fields are implicitly public, static and final, i.e., constants
* An interface can extend more than one interface and a class can implement more than one interface. This can be used to simulate multiple inheritance in Java
* A class can extend from only one class but can implement any number of interfaces
* The implements keyword is used to implement an interface. The classes implementing an interface must implement all the specified methods. Otherwise, they should be made abstract
* An interface creates a type. Hence, its reference can be used to refer to the objects of the classes which implement that interface. This leads to dynamic binding
* Since Java 8, an interface can also have default and static methods.\*\*

**Object class in Java:**

Object class is present in java.lang package and is the implicit super class of all classes. Every class in Java is directly or indirectly derived from the Object class.



Some of the commonly used methods of the Object class are:

* **equals()** - By default, equals() uses memory address to compare objects for equality (just like ==)

**\*Note**: While comparing String objects for equality, equals() method should be used since it is already overridden to compare the values.

* **hashCode()** - By default, it derives the hash value based on the memory address of the object being used. hashCode() uses an object's data to generate a hash value, which should be a 32-bit integer
* **toString()** - The **toString()** method returns a textual representation of an object.
  + By default, it returns a string consisting of the name of the object's class, the '@' character, and the unsigned hexadecimal representation of the hash code of the object. E.g. - Food@af7d0676.
  + It should be overridden to provide a meaningful textual representation. The returned text should be concise, easy to read and informative.

**Wrapper Classes in Java:**

A Wrapper class is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and, in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.

To convert data types into objects and to inherit the Object class, Java has **Boolean, Character, Integer, Long, Float** and **Double classes**which are called as **Wrapper Class**.

They belong to the **java.lang package** as part of the Java library.

**Need:**

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as [ArrayList](https://www.geeksforgeeks.org/arraylist-in-java/) and [Vector](https://www.geeksforgeeks.org/vector-vs-arraylist-java/), store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

**Primitive Data types and their Corresponding Wrapper class:**



Wrapper classes are final and immutable. Two concepts are there in the wrapper classes namely autoboxing and unboxing.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Autoboxing and Unboxing**

**Autoboxing:** Automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double etc.

**Unboxing:** It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

# **Exceptions in Java:**

**What is an Exception?**

An 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.



# **try:** The try block contains set of statements where an exception can occur.

**catch**: Catch block is used to handle the uncertain condition of try block. A try block is always followed by a catch block, which handles the exception that occurs in associated try block.

**finally**: It is executed after catch block. We basically use it to put some common code when there are multiple catch blocks.

**throw**: Throw keyword is used to transfer control from try block to catch block.

**throws**: Throws keyword is used for exception handling without try & catch block. It specifies the exceptions that a method can throw to the caller and does not handle itself.

try{

*//Statements that may cause an exception*

}

catch (ExceptionType e) ‏{

*//Error handling code*

}

finally {

*//Statements that must be executed*

}

**Packages:**

Packages help in organizing our Java files. It is a mechanism of grouping similar types of classes and interfaces collectively based on functionality.

Only public classes can be imported into different packages.

Packages can be divided into two categories:

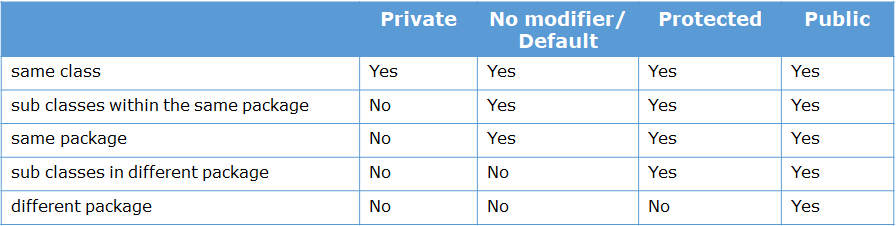
* Built-in Packages
* User-Defined Packages

**Built-in Packages**: Built-in packages or predefined packages are those that come along as a part of JDK (Java Development Kit). They consist of a huge number of predefined classes and interfaces that are part of Java API. Some of the commonly used built-in packages are java.lang, java.io, java.util, etc.

**User-Defined Packages**: User-defined packages are those which are developed by users in order to group related classes, interfaces and sub-packages.

* Packages help in organizing Java files by grouping similar Java files together.
* Every class is part of some package.
* When a package name is not specified, the classes are placed inside the default package.
* A class can have only one package statement but it can have more than one import package statements.
* The name of the package must be the same as the directory under which the file is saved.
* When importing another package, package declaration must be the first statement followed by package import.

The table given below explains the accessibility of the different access modifiers:



To summarize:

* Anything declared as public can be accessed from anywhere
* Anything declared as private can be accessed only within that class
* If access modifier is not mentioned, an element is accessible to all the classes in the same package
* Anything declared as protected can be accessible to all the classes inside the same package and to the sub classes present in a different package