What is Java?

Java is an [object-oriented](https://www.javatpoint.com/java-oops-concepts), class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

Object-Oriented Programming is a paradigm that provides many concepts, such as **inheritance**, **data binding**, **polymorphism**, etc.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995-.

Object-oriented programming is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields, and code, in the form of procedures. A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated.

Java Example

Let's have a quick look at Java programming example. A detailed description of Hello Java example is available in next page.

**class** Simple{

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

     System.out.println("Hello Java");

    }

}

**Platform**: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

## Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

## Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

#### **1) Standalone Application**

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

#### **2) Web Application**

An application that runs on the server side and creates a dynamic page is called a web application. Currently, [Servlet](https://www.javatpoint.com/servlet-tutorial), [JSP](https://www.javatpoint.com/jsp-tutorial), [Struts](https://www.javatpoint.com/struts-2-tutorial), [Spring](https://www.javatpoint.com/spring-tutorial), [Hibernate](https://www.javatpoint.com/hibernate-tutorial), [JSF](https://www.javatpoint.com/jsf-tutorial), etc. technologies are used for creating web applications in Java.

#### **3) Enterprise Application**

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, [EJB](https://www.javatpoint.com/ejb-tutorial) is used for creating enterprise applications.

Short for ***J***ava ***2*** Platform ***E***nterprise ***E***dition. J2EE is a [platform](https://www.webopedia.com/TERM/P/platform.html)-independent, [Java](https://www.webopedia.com/TERM/J/Java.html)-centric environment from Sun for developing, building and deploying Web-based enterprise applications online. The J2EE platform consists of a set of services, [APIs](https://www.webopedia.com/TERM/A/API.html), and [protocols](https://www.webopedia.com/TERM/P/protocol.html) that provide the functionality for developing multitiered, Web-based applications.

Some of the key features and services of J2EE:

* At the [client](https://www.webopedia.com/TERM/C/client.html) tier, J2EE supports pure [HTML](https://www.webopedia.com/TERM/H/HTML.html), as well as Java [applets](https://www.webopedia.com/TERM/A/applet.html) or applications. It relies on [Java Server Pages](https://www.webopedia.com/TERM/J/JSP.html) and [servlet](https://www.webopedia.com/TERM/S/servlet.html) code to create HTML or other formatted data for the client.
* [Enterprise JavaBeans](https://www.webopedia.com/TERM/E/Enterprise_JavaBeans.html) (EJBs) provide another layer where the platform's logic is stored. An EJB server provides functions such as threading, concurrency, security and memory management. These services are transparent to the author.
* [Java Database Connectivity](https://www.webopedia.com/TERM/J/JDBC.html) (JDBC), which is the Java equivalent to [ODBC](https://www.webopedia.com/TERM/O/ODBC.html), is the standard interface for Java databases.
* The Java servlet API enhances consistency for developers without requiring a [graphical user interface](https://www.webopedia.com/TERM/G/Graphical_User_Interface_GUI.html).

#### **4) Mobile Application**

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

# **What is the Difference between Web application and Enterprise application ?**

1.Web application is deployed in Web server which has web container  
Web application - JSP, Servlet, Java  
  
2.web apps are programs which created to run in a web browser.  
  
  
  
1.Enterprise App is deployed in Application server which has Web container, EJB container, set up for messaging services etc.  
EE app. - JSP, Servlet, Java, EJB, Web Services, JMS  
  
2.enterprise app is an integrated program which usually used in a large companies.

An enterprise application will contain enterprise beans(session/entity beans) and will run in a J2EE Container which provides transaction and security services to the beans and it will be named as .ear file. Whereas a web appliaction executes in a web-container like tomcat, contains servlets/jsps alone and named with an extension of .war.

# **Features of Java**

The primary objective of [Java programming](https://www.javatpoint.com/java-tutorial) language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java buzzwords.

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



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

# Why a Single Java Source File Can Not Have More Than One Public Class

According to Java standards and common practices, we should declare every class in its own source file. And even if we declare multiple classes in a single source file (.java), still each class will have its own class file after compilation. But the fact is that we can declare more than one class in a single source file with these constraints,

* Each source file should contain only one public class and the name of that public class should be similar to the name of the source file.
* If you are declaring a main method in your source file then main should lie in that public class.

If you are not following the first constraint then you will receive a compilation error saying “The public type A must be defined in its own file”. While if you are not following the second constraint you will receive an error “Error: Could not find or load main class User” after execution of the program, and if you try this in Eclipse, then you will not get the option to execute the program.

For every Class there will be a .(dot) file will be generated. If there are 4 class then there will be 4 . class files even if the 4 class are in single page.

Class A{

Public static void main(String args[])

}

Class B{

Public static void main(String args[])

}

Class C{

Public static void main(String args[])

}

Class D{

}

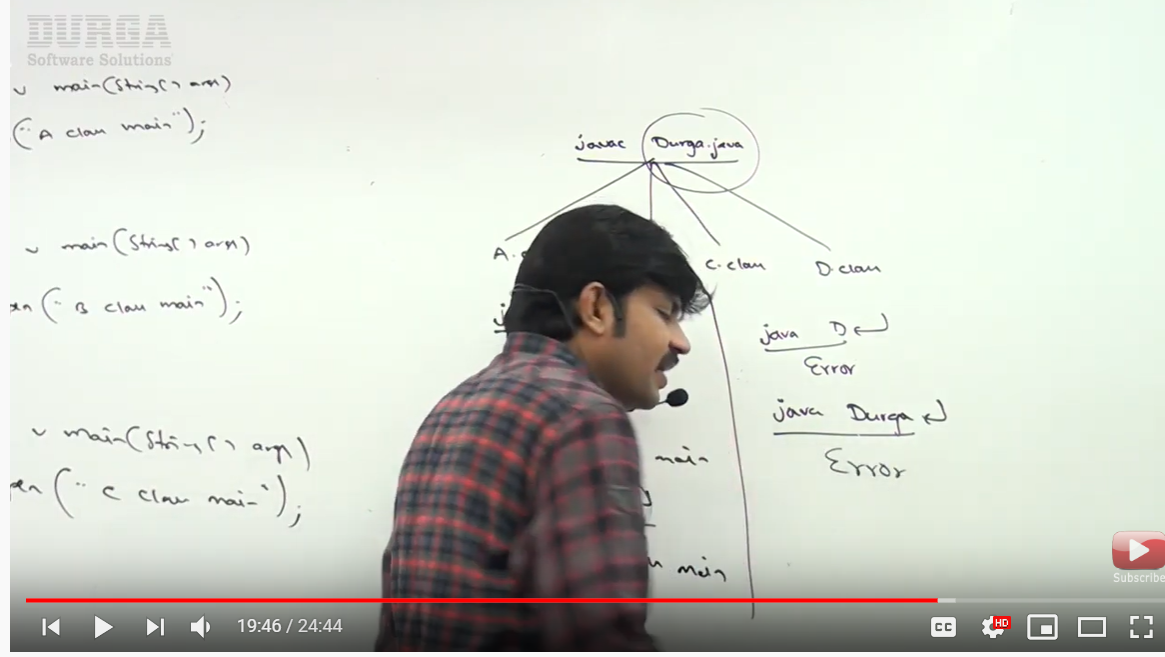
**Important Notes:**

Any source file can contain any number of java class and if Source file does not contain any public class then we can save the file name with any name. please note: we can make maximum 1 public class in any source file if we try to break rule we will get error saying that other files are public please make sure they are in other files.

But if the source file does not contain only 1 public class and it contain 3 main class then there will be 4 dot class file and based on what class we want to run that program will execute.

Like java A , java B .

But if we want to execute class D which does not contain main class then we will have error again. Saying no main method found.



### **PUBLIC STATIC VOID MAIN METHOD:**

### public

This is the access modifier of the main method. It has to be public so that java runtime can execute this method. Remember that if you make any method non-public then it’s not allowed to be executed by any program, there are some access restrictions applied. So it means that the main method has to be public. Let’s see what happens if we define the main method as non-public.

### static

When java runtime starts, there is no object of the class present. That’s why the main method has to be static so that JVM can load the class into memory and call the main method. If the main method won’t be static, JVM would not be able to call it because there is no object of the class is present. Let’s see what happens when we remove static from java main method.

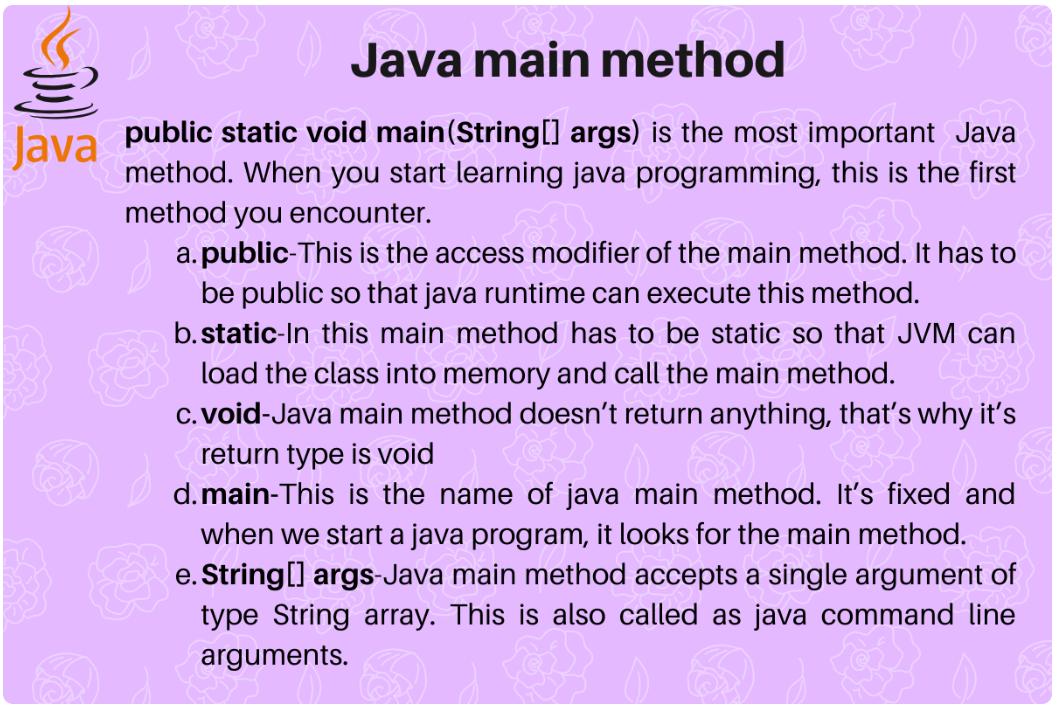
### void

Java programming mandates that every method provide the return type. Java main method doesn’t return anything, that’s why it’s return type is void. This has been done to keep things simple because once the main method is finished executing, java program terminates. So there is no point in returning anything, there is nothing that can be done for the returned object by JVM. If we try to return something from the main method, it will give compilation error as an unexpected return value. For example, if we have the main method like below.

### main

This is the name of java main method. It’s fixed and when we start a java program, it looks for the main method. For example, if we have a class like below.

### String[] args

Java main method accepts a single argument of type String array. This is also called as java command line arguments. Let’s have a look at the example of using java command line arguments

### **Object-oriented**

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

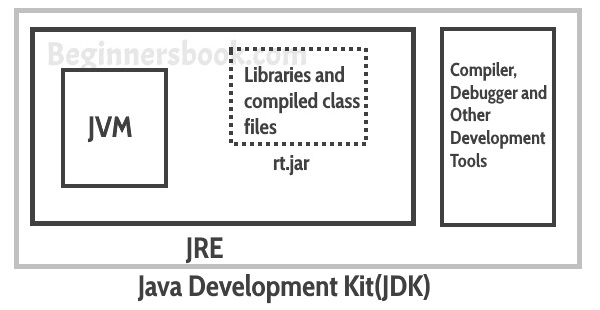
Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

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

**Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.

**Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.



## What happens at compile time?

At compile time, java file is compiled by Java Compiler (It does not interact with OS) and converts the java code into bytecode.





## What happens at runtime?

At runtime, following steps are performed:



|  |
| --- |
| **Classloader:**is the subsystem of JVM that is used to load class files. |
| **Bytecode Verifier:**checks the code fragments for illegal code that can violate access right to objects. |
| **Interpreter:**read bytecode stream then execute the instructions. |

**Date Types**

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

**Rule of Using package and improt in Java**

packagexyzpackage;

importabcpackage.\*;

publicclassTest {

}

How to access variable of another class in java ??

<https://stackoverflow.com/questions/1022880/accessing-a-variable-from-another-class>

You could make the variables public fields:

public int width;

public int height;

DrawFrame() {

this.width = 400;

this.height = 400;

}

You could then access the variables like so:

DrawFrame frame = new DrawFrame();

int theWidth = frame.width;

int theHeight = frame.height;

A better solution, however, would be to make the variables private fields add two accessor methods to your class, keeping the data in the DrawFrame class encapsulated:

private int width;

private int height;

DrawFrame() {

this.width = 400;

this.height = 400;

}

public int getWidth() {

return this.width;

}

public int getHeight() {

return this.height;

}

Then you can get the width/height like so:

DrawFrame frame = new DrawFrame();

int theWidth = frame.getWidth();

int theHeight = frame.getHeight();

I strongly suggest you use the latter method.

**2nd Way**

**package** accessModifier;

**class** AccessVariableFromAnotherClass{

**int** a = 10;

**static** **int** *b* = 20;

**final** **int** c = 30;

}

**public** **class** MainMethodClassForAcessVariable{

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

AccessVariableFromAnotherClass avfac = **new** AccessVariableFromAnotherClass();

/\* Instance variable can access with object \*/

System.***out***.println("Access Instance Variable like this :"+avfac.a);

/\* we can access static variable from two types \*/

/\* 1. we can access static variable with the help of object \*/

System.***out***.println("Access Static Variable like this :" +" " +avfac.*b*);

/\* 2. we can access static variable with the help of classname \*/

System.***out***.println("Access Static Variable like this :" +" " +AccessVariableFromAnotherClass.*b*);

/\* we cannot access static variable directly \*/

//System.out.println(b);

/\* we can access final variable with object \*/

System.***out***.println("Access Final Variable like this :" +" " +avfac.c);

/\* we cannot access final variable directly \*/

//System.out.println(c);

}

}

### **Understanding Java Access Modifiers**

in case of the object oriented languages, the data and the operational logic in functions is bundled together as an object. This creates a way for **code reusability** and **security,**which are the two main concepts of encapsulation.

We use access modifiers to define access control for classes, methods and variables.

Error when we break access modifier rules;

Exceptionin thread "main"java.lang.Error: Unresolved compilation problem:

The method addTwoNumbers(int, int) from the type Additionisnot visible

at xyzpackage.Test.main(Test.java:12)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **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 accessible only within the class.

**Simple example of private access modifier**

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

1. **class** A{
2. **private** **int** data=40;
3. **private** **void** msg(){System.out.println("Hello java");}
4. }
6. **public** **class** Simple{
7. **public** **static** **void** main(String args[]){
8. A obj=**new** A();
9. System.out.println(obj.data);//Compile Time Error
10. obj.msg();//Compile Time Error
11. }
12. }

### **Role of Private Constructor**

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

1. **class** A{
2. **private** A(){}//private constructor
3. **void** msg(){System.out.println("Hello java");}
4. }
5. **public** **class** Simple{
6. **public** **static** **void** main(String args[]){
7. A obj=**new** A();//Compile Time Error
8. }
9. }

#### **Note: A class cannot be private or protected except nested class.**

### **2) Default**

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

**Example of default access modifier**

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

1. //save by A.java
2. **package** pack;
3. **class** A{
4. **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
9. **class** B{
10. **public** **static** **void** main(String args[]){
11. A obj = **new** A();//Compile Time Error
12. obj.msg();//Compile Time Error
13. }
14. }

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

### **3) Protected**

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

**Example of protected access modifier**

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

1. //save by A.java
2. **package** pack;
3. **public** **class** A{
4. **protected** **void** msg(){System.out.println("Hello");}
5. }
6. //save by B.java
7. **package** mypack;
8. **import** pack.\*;
10. **class** B **extends** A{
11. **public** **static** **void** main(String args[]){
12. B obj = **new** B();
13. obj.msg();
14. }
15. }

Output:Hello

### **4) Public**

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

**Example of public access modifier**

1. //save by A.java
3. **package** pack;
4. **public** **class** A{
5. **public** **void** msg(){System.out.println("Hello");}
6. }
7. //save by B.java
9. **package** mypack;
10. **import** pack.\*;
12. **class** B{
13. **public** **static** **void** main(String args[]){
14. A obj = **new** A();
15. obj.msg();
16. }
17. }

Output:Hello

### **Java Access Modifiers with Method Overriding**

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

1. **class** A{
2. **protected** **void** msg(){System.out.println("Hello java");}
3. }
5. **public** **class** Simple **extends** A{
6. **void** msg(){System.out.println("Hello java");}//C.T.Error
7. **public** **static** **void** main(String args[]){
8. Simple obj=**new** Simple();
9. obj.msg();
10. }
11. }

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

Parameter Working in Java

A *parameter*is a value that you can pass to a method in Java. Then the method can use the parameter as though it were a local variable initialized with the value of the variable passed to it by the calling method.

The guessing-game application has a method named getRandomNumber that returns a random number between 1 and 10:

public static intgetRandomNumber()

{

return (int)(Math.random() \* 10) + 1;

}

This method is useful, but it would be even more useful if you could tell it the range of numbers you want the random number to fall in. It would be nice to call the method like this to get a random number between 1 and 10:

int number = getRandomNumber(1, 10);

Then, if your program needs to roll dice, you could call the same method:

int number = getRandomNumber(1, 6);

Or, to pick a random card from a deck of 52 cards, you could call it like this:

int number = getRandomNumber(1, 52);

You wouldn’t have to start with 1, either. To get a random number between 50 and 100, you’d call the method like this:

## Declaring parameters

A method that accepts parameters must list the parameters in the method declaration. The parameters are placed in a *parameter list*inside the parentheses that follow the method name. For each parameter used by the method, you list the parameter type followed by the parameter name. If you need more than one parameter, you separate the parameters with commas.

Here’s a version of the getRandomNumber method that accepts parameters:

public static intgetRandomNumber(int min, int max)

{

return (int)(Math.random()

\* (max – min + 1)) + min;

}

Here the method uses two parameters, both of type int, named min and max. Then, within the body of the method, these parameters can be used as though they were local variables.

int min = 1;

int max = 10;

int number = getRandomNumber(min, max);

## Understanding pass-by-value

When Java passes a variable to a method via a parameter, the method itself receives a copy of the variable’s value, not the variable itself. This copy is called a *pass-by-value,* and it has an important consequence: If a method changes the value it receives as a parameter, that change is *not*reflected in the original variable that was passed to the method. This program can help clear this up:

public class ChangeParameters

{

public static void main(String[] args)

{

int number = 1;

tryToChangeNumber(number);

System.out.println(number);

}

public static void tryToChangeNumber(inti)

{

i = 2;

}

}

Here a variable named number is set to 1 and then passed to the method named tryToChangeNumber. This method receives the variable as a parameter named i and then sets the value of i to 2. Meanwhile, back in the main method, println is used to print the value of number after the tryToChangeNumber method returns.

Because tryToChangeNumber gets only a copy of number, not the number variable itself, this program displays the following on the console: 1.

The key point is this: Even though the tryToChangeNumber method changes the value of its parameter, that change has no effect on the original variable that was passed to the method.

### **Coupling**

Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other. If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field. You can use interfaces for the weaker coupling because there is no concrete implementation.

### **Cohesion**

Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts. The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

**Why we are using getter and setter?**

By using getter and setter, the programmer can control how their important variables are accessed and updated, such as changing value of a variable within a specified range. Consider the following code of a setter method:

By using getter and setter we using the encapsulation concept of OOPs system where user decide what data must be show at what level.

# Getters & Setters

Getters and Setters are used to effectively protect your data, particularly when creating classes. For each instance variable, a getter method returns its value while a setter method sets or updates its value. Getters and setters are also known as accessors and mutators, respectively.

By convention, getters start with get, followed by the variable name, with the first letter of the variable name capitalized. Setters start with set, followed by the variable name, with the first letter of the variable name capitalized.

**Example:**

publicclassVehicle{

private String color;

// Getter

public String getColor(){

return color;

}

// Setter

publicvoidsetColor(String c){

this.color= c;

}

}

## What is Inheritance?

## <http://www.codespaghetti.com/inheritance-interview-questions/>

**Inheritance** is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

Extends Keyword is used to create the inheritance in java.

Class B{

}

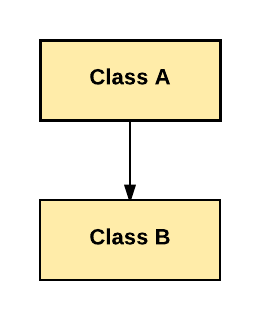
Class A extends B{ }

* Inheritance means one object acquires all the properties and behaviors of parent object.
* Inheritance is a compile-time mechanism.
* A super-class can have any number of subclasses. But a subclass can have only one superclass.
* The **extends keyword** indicates that you are making a new class that derives from an existing class.
* The superclass and subclass have **“is-a”** relationship between them.
* Inheritance is used For **Method Overriding** and For **Code Reusability.**

Types of Inheritance

### Single Inheritance:

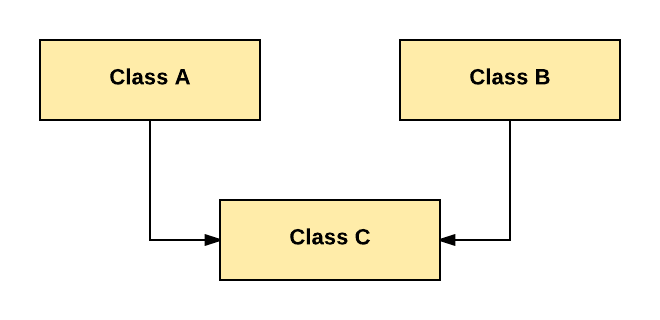
In Single Inheritance one class extends another class (one class only).

[](https://www.guru99.com/images/java/single_inheritance.png)Single Inheritance

In above diagram, Class B extends only Class A. Class A is a super class and Class B is a Sub-class.

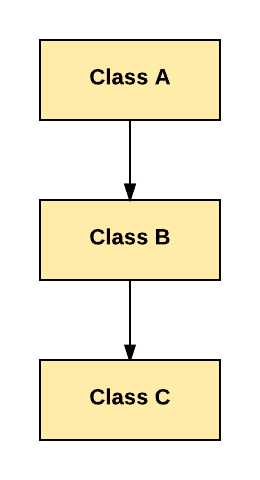
### Multiple Inheritance:

In Multiple Inheritance, one class extending more than one class. Java does not support multiple inheritance.

[](https://www.guru99.com/images/java/multiple.png)Multiple Inheritance

### Multilevel Inheritance:

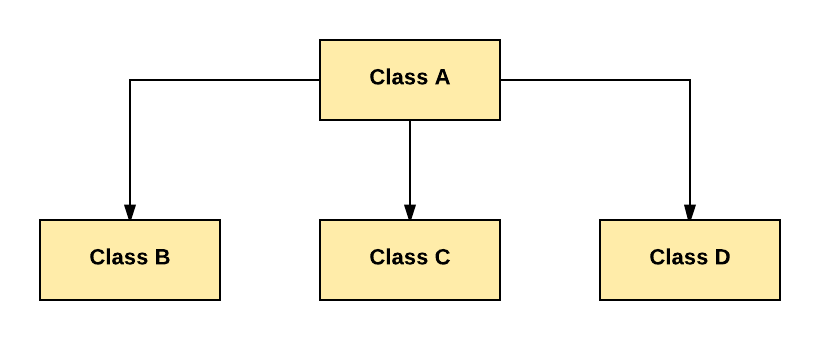
In Multilevel Inheritance, one class can inherit from a derived class. Hence, the derived class becomes the base class for the new class.

[](https://www.guru99.com/images/java/multilevel.png)Multilevel Inheritance

As per shown in diagram Class C is subclass of B and B is a of subclass Class A.

### Hierarchical Inheritance:

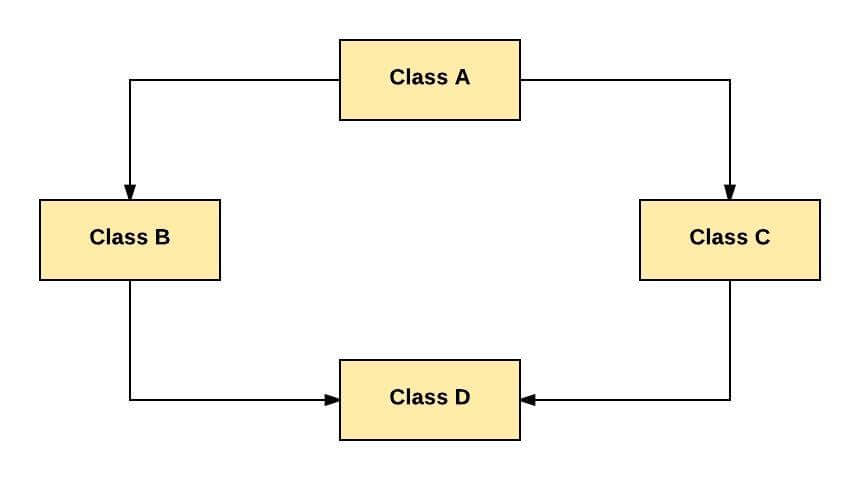
In Hierarchical Inheritance, one class is inherited by many sub classes.

[](https://www.guru99.com/images/java/hierarchy.png)Hierarchical Inheritance

As per above example, Class B, C, and D inherit the same class A.

### Hybrid Inheritance:

Hybrid inheritance is a combination of Single and Multiple inheritance.

[](https://www.guru99.com/images/java/hybrid.jpeg)Hybrid Inheritance

As per above example, all the public and protected members of Class A are inherited into Class D, first via Class B and secondly via Class C.

**Note:** Java doesn't support hybrid/Multiple inheritance.

## Inheritance in Java

**Java Inheritance** is a mechanism in which one class acquires the property of another class. In Java, when an "Is-A" relationship exists between two classes, we use Inheritance. The parent class called a super class and the inherited class called as a sub class. The keyword extends is used by the sub class to inherit the features of super class. Inheritance is important since it leads to the reusability of code.

**Java Inheritance Syntax:**

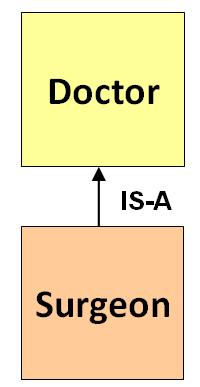
class subClass extends superClass

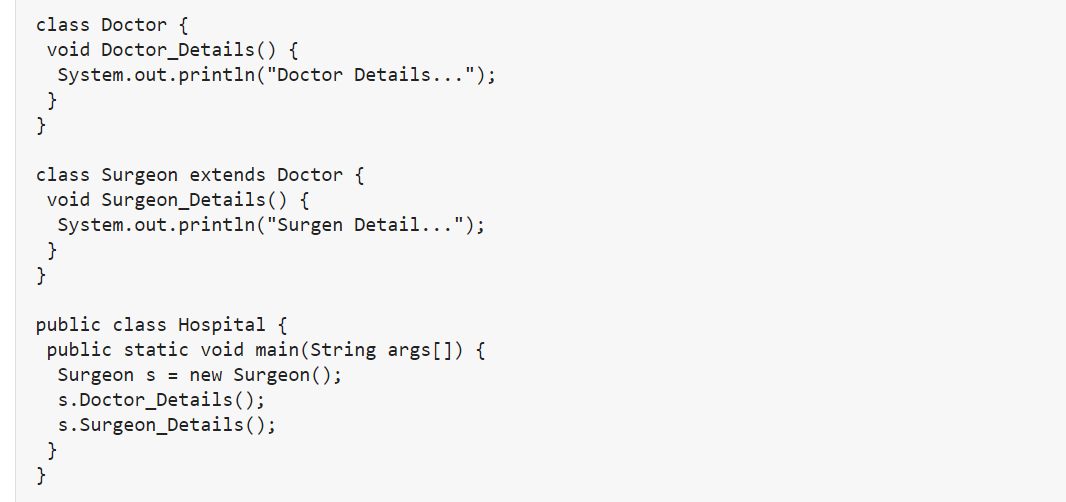
{

//methods and fields

}

## Java Inheritance Example

[](https://www.guru99.com/images/uploads/2012/07/java-inheritance.jpg)



## Super Keyword

The super keyword is similar to "this" keyword.

The keyword super can be used to access any data member or methods of the parent class.

Super keyword can be used at variable, method and constructor level.

**Syntax:**

super.<method-name>();

#### Q6. Why cannot private method be overridden?

**Answer:**  
The subclass did not inherit the properties of a private method and therefore it cannot be overridden. Moreover, this kind of a method is invisible to all the entities operating outside the class and the call to it is taken care at the compile time itself by making use of Type.

#### Can subclass inherit static members?

**Answer:**  
No, inheritance of static members is not possible but subclass, as well as the superclass, can both consist of the static method with a common signature. The subclass will have consisted of the properties of the static member of the superclass.

# Multiple inheritance by Interface in Java.

An interface contains variables and methods like a class but the methods in an interface are abstract by default unlike a class. Multiple inheritance by interface occurs if a class implements multiple interfaces or also if an interface itself extends multiple interfaces.

A program that demonstrates multiple inheritance by interface in Java is given as follows:

## Output

Animal is eating

Animal is travelling

Now let us understand the above program.

The interface AnimalEat and AnimalTravel have one abstract method each i.e. eat() and travel(). The class Animal implements the interfaces AnimalEat and AnimalTravel. A code snippet which demonstrates this is as follows:

interface AnimalEat {

   void eat();

}

interface AnimalTravel {

   void travel();

}

class Animal implements AnimalEat, AnimalTravel {

   public void eat() {

      System.out.println("Animal is eating");

   }

   public void travel() {

      System.out.println("Animal is travelling");

   }

}

In the method main() in class Demo, an object a of class Animal is created. Then the methods eat() and travel() are called. A code snippet which demonstrates this is as follows:

public class Demo {

   public static void main(String args[]) {

      Animal a = new Animal();

      a.eat();

      a.travel();

   }

}

### **What is Method Overriding And Method Hiding in Java?**

### **Method Overriding:**

An instance method in a subclass with the same signature (name, number and the type of its parameters) and return type as an instance method in the superclass *overrides* the superclass's method.

The overriding method has the same name, number and type of parameters, and return type as the method that it overrides.

An overriding method can also return a subtype of the type returned by the overridden method. This subtype is called a *covariant return type*.

When overriding a method, you might want to use the @Override annotation that instructs the compiler that you intend to override a method in the superclass.

If, for some reason, the compiler detects that the method does not exist in one of the superclasses, then it will generate an error.

**In**[**Java**](https://www.thoughtco.com/what-is-java-2034117)**, a method signature is part of the method declaration. It's the combination of the method name and the**[**parameter**](https://www.thoughtco.com/parameter-2034268)**list.**

### **Method Hiding:**

If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass ***hides*** the one in the superclass.

The distinction between hiding a static method and overriding an instance method has important implications:

* The version of the overridden instance method that gets invoked is the one in the subclass.
* The version of the hidden static method that gets invoked depends on whether it is invoked from the superclass or the subclass.

Consider an example that contains two classes. The first is Animal, which contains one instance method and one static method:

public class Animal {

public static void testClassMethod() {

System.out.println("The static method in Animal");

}

public void testInstanceMethod() {

System.out.println("The instance method in Animal");

}

}

The second class, a subclass of Animal, is called Cat:

public class Cat extends Animal {

public static void testClassMethod() {

System.out.println("The static method in Cat");

}

public void testInstanceMethod() {

System.out.println("The instance method in Cat");

}

public static void main(String[] args) {

Cat myCat = new Cat();

Animal myAnimal = myCat;

Animal.testClassMethod();

myAnimal.testInstanceMethod();

}

}

The Cat class overrides the instance method in Animal and hides the static method in Animal. The main method in this class creates an instance of Cat and invokes testClassMethod() on the class and testInstanceMethod() on the instance.

The output from this program is as follows:

The static method in Animal  
The instance method in Cat

As promised, the version of the hidden static method that gets invoked is the one in the superclass, and the version of the overridden instance method that gets invoked is the one in the subclass.

|  |  |  |
| --- | --- | --- |
| Defining a Method with the Same Signature as a Superclass's Method | | |
|  | **Superclass Instance Method** | **Superclass Static Method** |
| **Subclass Instance Method** | Overrides | Generates a compile-time error |
| **Subclass Static Method** | Generates a compile-time error | Hides |

**Note:** In a subclass, you can overload the methods inherited from the superclass.

Such overloaded methods neither hide nor override the superclass instance methods—they are new methods, unique to the subclass.

### **Question: Super() vs This() in Java**

**Super()**

* Super keyword is used to call constructor in the super class.
* Super always refers to the parent of the current class
* Super allows you to access public/protected method/attributes of parent class. You cannot see the parent's private method/attributes.
* Super allows access to constructors from within the class' constructors only.

**this()**

* this refers to a reference of the current class.
* this allows access methods/attributes of the current class (including its own private methods/attributes).
* this is used to access the methods and fields of the current object. For this reason, it has no meaning in static methods, for example. this keyword use to call constructor in the same class (other overloaded constructor)

**Is a return type different, when overriding in Java?**

By overriding a method, we mean implementing such a method in child class (overriding method), which is already implemented in a parent class (overridden method).

While overriding, it is necessary for the return type of the overriding method to be either:

* Exactly the same as return type of the overridden method.

OR

* The return type can be a sub-class of the return type of overridden method. This is facility is called covariant return type and was added in Java 5.0

# Covariant return types in Java

Covariant simply means return on class reference or its child class reference.

Covariant return type refers to return type of an overriding method. It allows to narrow down return type of an overridden method without any need to cast the type or check the return type. Covariant return type works only for non-primitive return types.

From Java 5 onwards, we can override a method by changing its return type only by abiding the condition that return type is a subclass of that of overridden method return type.

Following example showcases the same.

class SuperClass {

   SuperClass get() {

      System.out.println("SuperClass");

      return this;

   }

}

public class Tester extends SuperClass {

   Tester get() {

      System.out.println("SubClass");

      return this;

   }

   public static void main(String[] args) {

      SuperClass tester = new Tester();

      tester.get();

   }

}

## ****Hierarchy Example****

Let’s take this example.

**Food -> Fruit -> Apple, Orange**

Food is the interface which is at the topmost level

|  |  |
| --- | --- |
| 1  2  3  4 | public interface Food {     public float getTotalCalories();     public String getOrigin();  } |

The fruit is the abstract class

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public abstract class Fruit implements Food {  public float getTotalCalories(){        return 0.50f;  }     public String getOrigin();  } |