1. **FileWriter**: FileWriter is the simplest way to write a file in Java. It provides overloaded write method to write int, byte array, and String to the File. You can also write part of the String or byte array using FileWriter. FileWriter writes directly into Files and should be used only when the number of writes is less.
2. **BufferedWriter**: BufferedWriter is almost similar to FileWriter but it uses internal buffer to write data into File. So if the number of write operations is more, the actual IO operations are less and performance is better. You should use BufferedWriter when the number of write operations is more.
3. **FileOutputStream**: FileWriter and BufferedWriter are meant to write text to the file but when you need raw stream data to be written into file, you should use FileOutputStream to write file in java.
4. **Files**: Java 7 introduced Files utility class and we can write a file using its write function. Internally it’s using OutputStream to write byte array into file.

package com.journaldev.files;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileOutputStream;

import java.io.FileWriter;

import java.io.IOException;

import java.io.OutputStream;

import java.nio.file.Files;

import java.nio.file.Paths;

public class WriteFile {

/\*\*

\* This class shows how to write file in java

\* @param args

\* @throws IOException

\*/

public static void main(String[] args) {

String data = "I will write this String to File in Java";

int noOfLines = 10000;

writeUsingFileWriter(data);

writeUsingBufferedWriter(data, noOfLines);

writeUsingFiles(data);

writeUsingOutputStream(data);

System.out.println("DONE");

}

/\*\*

\* Use Streams when you are dealing with raw data

\* @param data

\*/

private static void writeUsingOutputStream(String data) {

OutputStream os = null;

try {

os = new FileOutputStream(new File("/Users/pankaj/os.txt"));

os.write(data.getBytes(), 0, data.length());

} catch (IOException e) {

e.printStackTrace();

}finally{

try {

os.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

/\*\*

\* Use Files class from Java 1.7 to write files, internally uses OutputStream

\* @param data

\*/

private static void writeUsingFiles(String data) {

try {

Files.write(Paths.get("/Users/pankaj/files.txt"), data.getBytes());

} catch (IOException e) {

e.printStackTrace();

}

}

/\*\*

\* Use BufferedWriter when number of write operations are more

\* It uses internal buffer to reduce real IO operations and saves time

\* @param data

\* @param noOfLines

\*/

private static void writeUsingBufferedWriter(String data, int noOfLines) {

File file = new File("/Users/pankaj/BufferedWriter.txt");

FileWriter fr = null;

BufferedWriter br = null;

String dataWithNewLine=data+System.getProperty("line.separator");

try{

fr = new FileWriter(file);

br = new BufferedWriter(fr);

for(int i = noOfLines; i>0; i--){

br.write(dataWithNewLine);

}

} catch (IOException e) {

e.printStackTrace();

}finally{

try {

br.close();

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

/\*\*

\* Use FileWriter when number of write operations are less

\* @param data

\*/

private static void writeUsingFileWriter(String data) {

File file = new File("/Users/pankaj/FileWriter.txt");

FileWriter fr = null;

try {

fr = new FileWriter(file);

fr.write(data);

} catch (IOException e) {

e.printStackTrace();

}finally{

//close resources

try {

fr.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

Introduction

The **java.util.Scanner** class is a simple text scanner which can parse primitive types and strings using regular expressions.Following are the important points about Scanner −

* A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace.
* A scanning operation may block waiting for input.
* A Scanner is not safe for multithreaded use without external synchronization.

Class declaration

Following is the declaration for **java.util.Scanner** class −

public final class Scanner

extends Object

implements Iterator<String>

 By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

[**next →**](https://www.javatpoint.com/PrintStream-class)[**← prev**](https://www.javatpoint.com/Console-class)

# **Java Scanner**

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

## Java Scanner Class Declaration

1. **public** **final** **class** Scanner
2. **extends** Object
3. **implements** Iterator<String>

## How to get Java Scanner

To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

1. Scanner in = **new** Scanner(System.in);

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

1. Scanner in = **new** Scanner("Hello Javatpoint");

There are many ways to read a text file in java. A text file is made of characters, so we can use Reader classes. There are some utility classes too to read a text file in java.

1. Java read text file using [Files class](https://www.journaldev.com/17794/java-files-nio-files-class)
2. Read text file in java using [FileReader](https://www.journaldev.com/19115/java-filereader" \t "_blank)
3. Java read text file using [BufferedReader](https://www.journaldev.com/19879/java-bufferedreader" \t "_blank)
4. Using [Scanner class](https://www.journaldev.com/872/scanner-class-in-java) to read text file in java

Now let’s look at examples showing how to read a text file in java using these classes.

### Java read text file using java.nio.file.Files

We can use Files class to read all the contents of a file into a byte array. Files class also has a method to read all lines to a list of string. Files class is introduced in Java 7 and it’s good if you want to load all the file contents. You should use this method only when you are working on small files and you need all the file contents in memory.

String fileName = "/Users/pankaj/source.txt";

Path path = Paths.get(fileName);

byte[] bytes = Files.readAllBytes(path);

List<String> allLines = Files.readAllLines(path, StandardCharsets.UTF\_8);

### Read text file in java using java.io.FileReader

You can use FileReader to get the BufferedReader and then read files line by line. FileReader doesn’t support encoding and works with the system default encoding, so it’s not a very efficient way of reading a text file in java.

String fileName = "/Users/pankaj/source.txt";

File file = new File(fileName);

FileReader fr = new FileReader(file);

BufferedReader br = new BufferedReader(fr);

String line;

while((line = br.readLine()) != null){

//process the line

System.out.println(line);

}

### Java read text file using java.io.BufferedReader

BufferedReader is good if you want to read file line by line and process on them. It’s good for processing the large file and it supports encoding also.

BufferedReader is synchronized, so read operations on a BufferedReader can safely be done from multiple threads. BufferedReader default buffer size is 8KB.

String fileName = "/Users/pankaj/source.txt";

File file = new File(fileName);

FileInputStream fis = new FileInputStream(file);

InputStreamReader isr = new InputStreamReader(fis, cs);

BufferedReader br = new BufferedReader(isr);

String line;

while((line = br.readLine()) != null){

//process the line

System.out.println(line);

}

br.close();

### Using scanner to read text file in java

If you want to read file line by line or based on some [java regular expression](https://www.journaldev.com/634/regular-expression-in-java-regex-example), Scanner is the class to use.

Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods. The scanner class is not synchronized and hence not thread safe.

Path path = Paths.get(fileName);

Scanner scanner = new Scanner(path);

System.out.println("Read text file using Scanner");

//read line by line

while(scanner.hasNextLine()){

//process each line

String line = scanner.nextLine();

System.out.println(line);

}

scanner.close();

## Read a String from console input in Java

In this Java Tutorial, we shall useScanner scanner = new Scanner(System. in); to read a String from console input in Java.

ReadString

|  |
| --- |
| import java.util.Scanner;    /\*\*  \* An example program to read a String from console input in Java  \*/  public class ReadString {        public static void main(String[] args) {          System.out.print("Enter a string : ");          Scanner scanner = new Scanner(System. in);          String inputString = scanner. nextLine();          System.out.println("String read from console is : \n"+inputString);      }  } |

When the program is run, the execution waits after printing “Enter a string : “, where you would enter a string like “hello world”. The program prints the string read from the console input in the next step. The whole output in the console would be as shown in the following :

Output

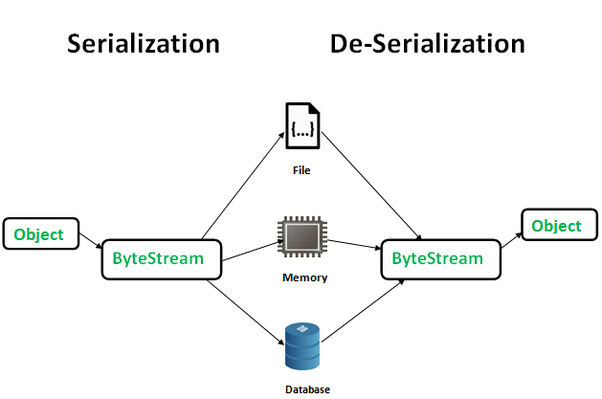
|  |
| --- |
| Enter a string : hello world  String read from console is :  hello world |

### System.in

In the context of reading something from the console, [System](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html) class provides a means to access standard input through one of its fields, [in](https://docs.oracle.com/javase/8/docs/api/java/lang/System.html#in). ‘in’ field is a Stream (to be specific, its a InputStream), which is declared public static and final. Hence, one can use ‘in’ directly without any initialization. Typically, this InputStream corresponds to keyboard input or another input source specified by the host environment or user.

**Serialization and Deserialization in Java with Example**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.



The byte stream created is platform independent. So, the object serialized on one platform can be deserialized on a different platform.

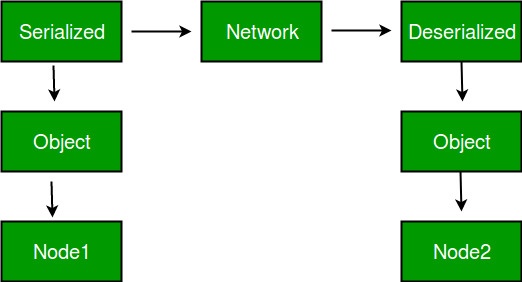
To make a Java object serializable we implement the [**http://java.io**](http://java.io/)**.Serializable** interface.  
The ObjectOutputStream class contains **writeObject()** method for serializing an Object.

1. **public** final **void** writeObject(Object obj)
2. throws IOException

The ObjectInputStream class contains **readObject()** method for deserializing an object.

1. **public** final Object readObject()
2. throws IOException,
3. ClassNotFoundException

**Advantages of Serialization**  
1. To save/persist state of an object.  
2. To travel an object across a network.



Only the objects of those classes can be serialized which are implementing [**http://java.io**](http://java.io/)**.Serializable** interface.  
Serializable is a **marker interface** (has no data member and method). It is used to “mark” java classes so that objects of these classes may get certain capability. Other examples of marker interfaces are:- Cloneable and Remote.

**Points to remember**  
1. If a parent class has implemented Serializable interface then child class doesn’t need to implement it but vice-versa is not true.  
2. Only non-static data members are saved via Serialization process.  
3. Static data members and transient data members are not saved via Serialization [http://process.So](http://process.so/), if you don’t want to save value of a non-static data member then make it transient.  
4. Constructor of object is never called when an object is deserialized.  
5. Associated objects must be implementing Serializable interface.  
Example :

1. **class** A implements Serializable{
3. // B also implements Serializable
4. // interface.
5. B ob=**new** B();
6. }

**SerialVersionUID**  
The Serialization runtime associates a version number with each Serializable class called a SerialVersionUID, which is used during Deserialization to verify that sender and reciever of a serialized object have loaded classes for that object which are compatible with respect to serialization. If the reciever has loaded a class for the object that has different UID than that of corresponding sender’s class, the Deserialization will result in an **InvalidClassException**. A Serializable class can declare its own UID explicitly by declaring a field name.  
It must be static, final and of type long.  
i.e- ANY-ACCESS-MODIFIER static final long serialVersionUID=42L;

If a serializable class doesn’t explicitly declare a serialVersionUID, then the serialization runtime will calculate a default one for that class based on various aspects of class, as described in Java Object Serialization Specification. However it is strongly recommended that all serializable classes explicitly declare serialVersionUID value, since its computation is highly sensitive to class details that may vary depending on compiler implementations, any change in class or using different id may affect the serialized data.

It is also recommended to use private modifier for UID since it is not useful as inherited member.

**serialver**  
The serialver is a tool that comes with JDK. It is used to get serialVersionUID number for Java classes.  
You can run the following command to get serialVersionUID

serialver [-classpath classpath] [-show] [classname…]

=>**Serialization :**

* The process of ***writing a state of an Object***to a file is called ***Serialization***
* In other words, the ***process of saving an Object’s state to a file*** is called ***Serialization***

**De-Serialization :**

* The ***process of reading a state of an Object from a file*** is called ***De-Serialization***
* Retrieving those saved bytes into the form of original object.
* To prevent Serialization use keyword **Transient.**

# Abstract method in Java with examples

BY CHAITANYA SINGH | FILED UNDER: [OOPS CONCEPT](https://beginnersbook.com/category/oops-concept/)

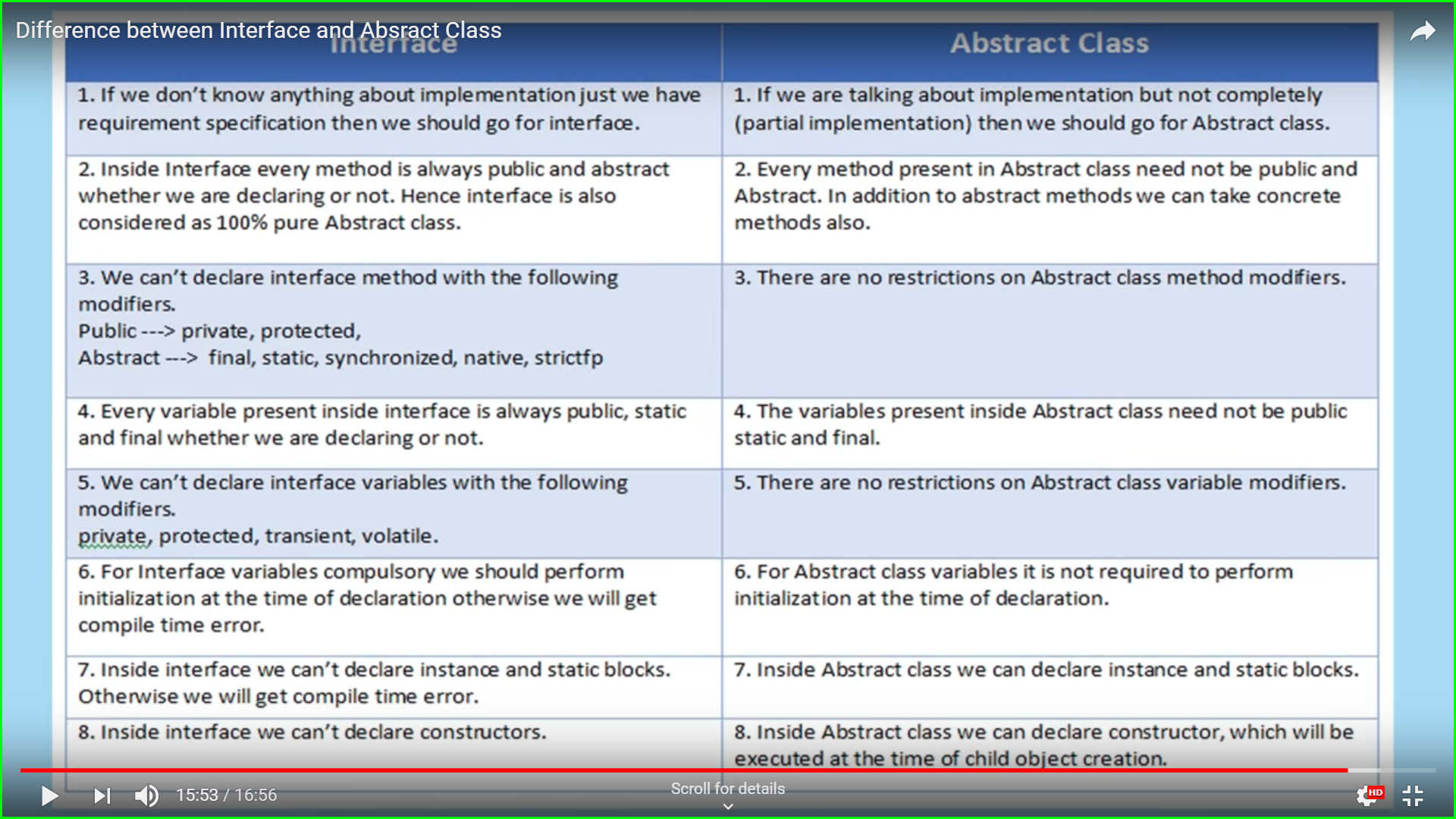
A method without body (no implementation) is known as abstract method. A method must always be declared in an abstract class, or in other words you can say that if a class has an abstract method, it should be declared abstract as well. In the last tutorial we discussed Abstract class, if you have not yet checked it out read it here: [Abstract class in Java](https://beginnersbook.com/2013/05/java-abstract-class-method/), before reading this guide.  
This is how an abstract method looks in java:

public abstract int myMethod(int n1, int n2);

As you see this has no body.

## Rules of Abstract Method

1. Abstract methods don’t have body, they just have method signature as shown above.  
2. If a class has an abstract method it should be declared abstract, the vice versa is not true, which means an abstract class doesn’t need to have an abstract method compulsory.  
3. If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.

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

This mechanism of deriving a new class from existing/old class is called “inheritance”.

The old class is known as “base” class, “super” class or “parent” class”; and the new class is known as “sub” class, “derived” class, or “child” class.

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

* For Method Overriding (so runtime polymorphism 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.

### **The syntax of Java Inheritance**

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

The inheritance allows subclasses to inherit all

properties (variables and methods) of their

parent classes. The different forms of

inheritance are:

. Single inheritance (only one super class)

. Multiple inheritance (several super classes)

. Hierarchical inheritance (one super class, many sub

classes)

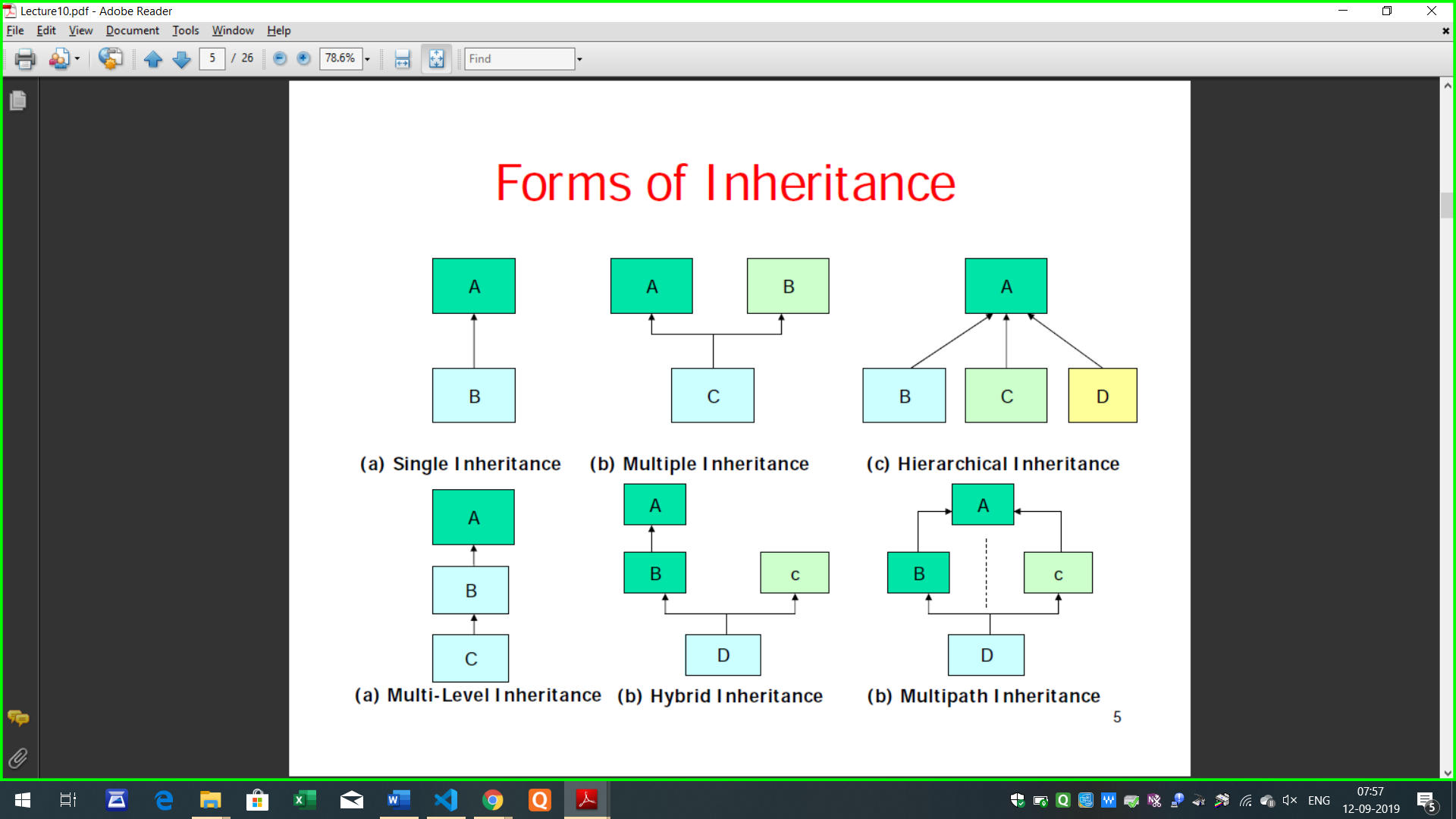
. Multi-Level inheritance (derived from a derived

class)

. Hybrid inheritance (more than two types)

. Multi-path inheritance (inheritance of some

properties from two sources).



[**next →**](https://www.javatpoint.com/aggregation-in-java)[**← prev**](https://www.javatpoint.com/this-keyword)

# **Inheritance in Java**

1. [Inheritance](https://www.javatpoint.com/inheritance-in-java)
2. [Types of Inheritance](https://www.javatpoint.com/inheritance-in-java#inheritancetypes)
3. [Why multiple inheritance is not possible in Java in case of class?](https://www.javatpoint.com/inheritance-in-java#inheritancenotmultiple)

**Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes 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 (so runtime polymorphism 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.

### **The syntax of Java Inheritance**

1. **class** Subclass-name **extends** Superclass-name
2. {
3. //methods and fields
4. }

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.

### **Java Inheritance Example**



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

1. **class** Employee{
2. **float** salary=40000;
3. }
4. **class** Programmer **extends** Employee{
5. **int** bonus=10000;
6. **public** **static** **void** main(String args[]){
7. Programmer p=**new** Programmer();
8. System.out.println("Programmer salary is:"+p.salary);
9. System.out.println("Bonus of Programmer is:"+p.bonus);
10. }
11. }

[**Test it Now**](http://www.javatpoint.com/opr/test.jsp?filename=Programmer)

Programmer salary is:40000.0

Bonus of programmer is:10000

In the above example, Programmer object can access the field of own class as well as of Employee class i.e. code reusability.

## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



#### **Note: Multiple inheritance is not supported in Java through class.**

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



## Single Inheritance Example

*File: TestInheritance.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** TestInheritance{
8. **public** **static** **void** main(String args[]){
9. Dog d=**new** Dog();
10. d.bark();
11. d.eat();
12. }}

Output:

barking...

eating...

## Multilevel Inheritance Example

*File: TestInheritance2.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** BabyDog **extends** Dog{
8. **void** weep(){System.out.println("weeping...");}
9. }
10. **class** TestInheritance2{
11. **public** **static** **void** main(String args[]){
12. BabyDog d=**new** BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}

Output:

weeping...

barking...

eating...

## Hierarchical Inheritance Example

*File: TestInheritance3.java*

1. **class** Animal{
2. **void** eat(){System.out.println("eating...");}
3. }
4. **class** Dog **extends** Animal{
5. **void** bark(){System.out.println("barking...");}
6. }
7. **class** Cat **extends** Animal{
8. **void** meow(){System.out.println("meowing...");}
9. }
10. **class** TestInheritance3{
11. **public** **static** **void** main(String args[]){
12. Cat c=**new** Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}

Output:

meowing...

eating...

## 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.

# Interface in java

[April 20, 2012](https://javabeginnerstutorial.com/core-java-tutorial/java-interface/)  [22 Comments](https://javabeginnerstutorial.com/core-java-tutorial/java-interface/#comments)  [J Singh](https://javabeginnerstutorial.com/author/admin/)

**Page Content**[[show](https://javabeginnerstutorial.com/core-java-tutorial/java-interface/)]

Defining a Contract means to create an Interface. This Contract states what a Class can do without forcing how it should do it. In this Java Interface Tutorial, we will talk about Interfaces. How to create one and rules applied to Interfaces.

## Declaring an Interface

Interfaces can be defined with the **Interface** keyword. After Java 8, Interface definition changes a lot.

## Java Interface Example

1. public interface MyInterface
2. {
3. int i=0;
4. public void Height(int height);
5. public abstract void setHeight();
6. }

## Java 8 Interface Example

1. package com.jbt;
2. public interface Interface\_JAVA8 {
3. public void method();
4. public abstract void method1();
5. public default void method2() {
6. }
7. public static void method3() {
8. }
9. }

Java 8 introduces a concept of Default Method. Now an Interface can contain Method body if it is not abstract. A  method inside Interface can contain **either of** “abstract”, “static” or “default” modifier.

## Java 9 Interface Private Method

In Java 9 you are allowed to have private methods in Interfaces.

## Rules for Declaring Interface

There are some rules that need to be followed by Interface.

* All interface Methods are implicitly public and abstract. Even if you use keyword it will not create the problem as you can see in the second Method declaration. (Before Java 8)
* Interfaces can declare only Constant. Instance variables are not allowed. This means all variables inside the Interface must be public, static, final. Variables inside Interface are implicitly public static final.
* Interface Methods cannot be static. (Before Java 8)
* Interface Methods cannot be final, strictfp or native.
* The Interface can extend one or more other Interface. Note: The Interface can only extend another interface.

## Rules for Declaring Interface(JAVA 8)

Interface logic has been changed in Java 8. Hence some of the above logic doesn’t apply on Interface after Java 8.

* All Interface Method can have either “abstract”, “static” or “default” modifier.
* Interface methods can have a body if static or default modifier is used against the method.

## Interface vs Abstract Class

An interface is like having a 100% Abstract Class. Interfaces can not have non-abstract Methods while abstract Classes can. A Class can implement more than one Interface while it can extend only one Class. As abstract Classes comes in the hierarchy of Classes, they can extend other Classes while Interface can only extend Interfaces.

## Use Interface in Class

How can we take advantage of an Interface after creating it in Java? To take advantage we need to implement our Class with a given Interface. The Implement keyword can be used for this purpose.

* An interface is 100% abstract class(***Implicitly***). After Java 8 it doesn’t hold true.
* Interfaces can be implemented by any class from any inheritance tree.
* All methods in Interfaces are abstract. (In Java 8 either abstract/ static / default)
* An interface can have **constants**, these constants are public, static and final(***Implicitly***).
* Interface methods are implicitly ***public & abstract. (Before Java 8)***
* An interface can also have private methods. (Java 9)
* A class implementing an interface can also be an abstract class.
* An abstract class which is implementing an interface need not implement all abstract method.
* A class can *Implement more than one* Interface.
* Interfaces can not extend a class or implement an Interface.
* An interface can extend another Interface.
* A non-abstract class which is implementing an Interface needs to follow some rules.
  1. This class needs to provide the concrete implementation of all abstract method.
  2. All rules of Overriding needs to be followed.
  3. It must maintain the exact signature of a method.
* After Java 9 changes, Interfaces look a lot like Abstract Class, still, there are some differences.
  1. An abstract class can have variables with different modifiers which is not constant
  2. Methods in Abstract can have the different signature than just private or public

why-cant-we-instantiate-a-abstract-class-in-java?

1. If a class is marked with keyword abstract then it is called abstract class. It can NOT be instantiated by using new operator. But an abstract class can be used as the superclass reference for the subclass object
2. A class that is NOT an abstract class is called concrete class
3. There are so many abstract classes in the Java API and most of them include in the GUI Library and Collections Framework
4. An abstract class must be extended and an abstract method must be overridden
5. If a method is declared with the keyword abstract then it should NOT have body. Only it can have name, parameters and return type but NOT the body
6. If you have an abstract method then the class must be declared as abtracr as well i.e. a concrete class can NOT have abstract methods only an abstract class can have abstract methods
7. Along with the abstract methods an abstract class can also have instance members, instance initializer, constructors, static initializers and static members. Here the constructors and instance initializer of the abstract class would be invoked when the subclass of the abstract class will be instantiated and the static initializer of the abstract class will be invoked when the abstract class will be loaded by the JVM
8. If a concrete class extends an abstract class then it must override all of the abstract methods of the abstract superclass and this is called the implementation of an abstract class
9. For the partial implementation of an abstract class we can take an abstract subclass that can extend the abstract superclass. In this situation the subclass override all of the abstract methods of the abstract superclass is NOT necessary. The partial implementation of an abstract superclass or an interface can be done only by the abstract classes. A concrete class can NOT be used for the partial implementation of an abstract class because concrete classes can NOT have abstract methods
10. Abstract classes and abstract methods can NOT be declared with keyword final i.e. a class can be abstract or final similarly a method can be abstract of final
11. Abstract methods can NOT be declared with keyword private
12. Abstract methods can NOT be declared with keyword synchronized. Even it you need a Thread safe method then during the implementation of the abstract class (when the abstract method will be overridden by the conrete subclass) the overridden method can be marked with keyword synchronized
13. Abstract methods can NOT be marked with keyword static. Because static methods can NOT be overridden only they can be hidden by the subclass static methods
14. Abstract method can NOT be marked with keyword native
15. Abstract method can NOT be marked with keyword strictfp
16. So only you can use the access modifier (other then private), type, parameter with an abstract method
17. A conrete class can NOT have instance methods but an abstract class can have abstract methods, static members, instance members, static initializers, constructors and instance initializer
18. If an abstract class implements an interface or extends another abstract class then it is NOT necessary that it must override all of the abstract methods of the interface. But this is NOT same as with the concrete classes
19. It may also happen that an abstract class does NOT have any of the abstract methods
20. An abstract classs can NOT be instantiated by using new operator. Becuase an abstract may have abstract methods i.e. methods without any body (or implementation). Because an object can NOT have an abstract methods and JVM can NOT allocate memory of the abstract methods

rules-for-abstract-methods-and-abstract-classes

and can abstract classes inherit from other abastract classs

Any sub-class extending from an abstract class should either implement all the abstract methods of the super-class or the sub-class itself should be marked as abstract.