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

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

Constructors are used for initializing new objects. Fields are variables that provides the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

- methods operate on the internal state (fields or variable)of an object and the object-to-object communication is done via methods.

There are various types of classes that are used in real time applications such as [nested classes](https://www.geeksforgeeks.org/inner-class-java/), [anonymous classes](https://www.geeksforgeeks.org/anonymous-inner-class-java/), [lambda expressions](https://www.geeksforgeeks.org/lambda-expressions-java-8/).

**Difference between class and object**

the class is not shaded, because it represents a blueprint of an object rather than an object itself. In comparison, an object is shaded, indicating that the object exists and that you can use it.

|  |  |  |
| --- | --- | --- |
| **Object** | | **Class** |
| 1) | Object is an **instance** of a class. | | Class is a **blueprint or template** from which  objects are created. |
| 2) | Object is a **real world entity** such as pen, laptop, mobile, bed, keyboard, mouse, chair etc. | | Class is a **group of similar objects**. |
| 3) | Object is a **physical** entity. | | Class is a **logical** entity. |
| 4) | Object is created through **new keyword** mainly e.g. Student s1=new Student(); | | Class is declared using **class keyword**e.g. class Student{} |
| 5) | Object is created **many times** as per requirement. | | Class is declared **once**. |
| 6) | Object **allocates memory when it is created**. | | Class **doesn't allocated memory when it is created**. |
| 7) | There are **many ways to create object** in java such as new keyword, newInstance() method, clone() method, factory method and deserialization. | | There is only **one way to define class** in java using class keyword. |

-A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object.

-An object is nothing but a self-contained component which consists of methods and properties to make a particular type of data useful. Object determines the behavior of the class. When you send a message to an object, you are asking the object to invoke or execute one of its methods.

**Access specifiers for classes or interfaces in Java**

methods and data members of a class/interface can have one of the following four access specifiers. The access specifiers are listed according to their restrictiveness order.

1)private  
2)default(when no access specifier is specified)  
3)protected  
4) public

But, the classes and interfaces themselves can have only two access specifiers when declared outside any other class.  
1)public  
2) default (when no access specifier is specified)

We cannot declare class/interface with private or protected access specifiers. For example, following program fails in compilation.

|  |
| --- |
| //filename: Main.java  protected class Test {}    public class Main {    public static void main(String args[]) {      }  } |

Run on IDE

Note : Nested interfaces and classes can have all access specifiers.

**A class can contain any of the following variable types.**

* **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
* **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
* **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword.

# Java Access Modifiers

A *Java access modifier* specifies which classes can access a given class and its fields, constructors and methods. Access modifiers can be specified separately for a class, its constructors, fields and methods. Java access modifiers are also sometimes referred to in daily speech as *Java access specifiers*, but the correct name is Java access modifiers. Classes, fields, constructors and methods can have one of four different Java access modifiers:

* private
* default (package)
* protected
* public

Each of these Java access modifiers will be covered in the following sections.

Assigning an access modifier to a class, constructor, field or method is also sometimes referred to as "marking" that class, constructor, field or method as that which the access modifier specifies. For instance, assigning the Java access modifier public to a method would be referred to as marking the method as public.

## private Access Modifier

If a method or variable is marked as private (has the private access modifier assigned to it), then only code inside the same class can access the variable, or call the method. Code inside subclasses cannot access the variable or method, nor can code from any external class.

Classes cannot be marked with the private access modifier. Marking a class with the private access modifier would mean that no other class could access it, which means that you could not really use the class at all. Therefore the private access modifier is not allowed for classes.

Here is an example of assigning the private access modifier to a field:

public class Clock {

**private** long time = 0;

}

The member variable time has been marked as private. That means, that the member variable time inside the Clock class cannot be accessed from code outside the Clock class.

### Accessing private Fields via Accessor Methods

Fields are often declared private to control the access to them from the outside world. In some cases the fields are truly private, meaning they are only used internally in the class. In other cases the fields can be accessed via accessor methods (e.g. getters and setters). Here is an accessor method example:

public class Clock {

private long time = 0;

public long getTime() {

return this.time;

}

public void setTime(long theTime) {

this.time = theTime;

}

}

In the above example the two methods getTime() and setTime() can access the time member variable. The two methods are declared public, meaning they can be called from code anywhere in your application. The public Java access modifier is covered later in this text.

### private Constructors

If a constructor in a class is assigned the private Java access modifier, that means that the constructor cannot be called from anywhere outside the class. A private constructor can still get called from other constructors, or from static methods in the same class. Here is a Java class example illustrating that:

public class Clock {

private long time = 0;

private Clock(long time) {

this.time = time;

}

public Clock(long time, long timeOffset) {

this(time);

this.time += timeOffset;

}

public static Clock newClock() {

return new Clock(System.currentTimeMillis());

}

}

This version of the Clock class contains a private constructor and a public constructor. The privateconstructor is called from the public constructor (the statement this();). The private constructor is also called from the static method newClock().

The above example only serves to show you that a private constructor can be called from public constructors and from static methods inside the same class. Do not perceive the above example as an example of clever design in any way.

## default (package) Access Modifier

The default Java access modifier is declared by not writing any access modifier at all. The default access modifier means that code inside the class itself as well as code inside classes in the same package as this class, can access the class, field, constructor or method which the default access modifier is assigned to. Therefore, the default access modifier is also sometimes referred to as the package access modifier. If you don't know what a Java package is, I have explained that in my [**Java packages tutorial**](http://tutorials.jenkov.com/java/packages.html).

Subclasses cannot access methods and member variables (fields) in the superclass, if they these methods and fields are marked with the default access modifier, unless the subclass is located in the same package as the superclass.

Here is an default / package access modifier example:

public class Clock {

long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

The time field in the Clock class has no access modifier, which means that it is implicitly assigned the default / package access modifier. Therefore, the ClockReader class can read the time member variable of the Clock object, provided that ClockReader and Clock are located in the same Java package.

## protected Access Modifier

The protected access modifier provides the same access as the default access modifier, with the addition that subclasses can access protected methods and member variables (fields) of the superclass. This is true even if the subclass is not located in the same package as the superclass.

Here is a protected access modifier example:

public class Clock {

**protected** long time = 0; // time in milliseconds

}

public class SmartClock() extends Clock{

public long getTimeInSeconds() {

return this.time / 1000;

}

}

In the above example the subclass SmartClock has a method called getTimeInSeconds() which accesses the time variable of the superclass Clock. This is possible even if Clock and SmartClock are not located in the same package, because the time field is marked with the protected Java access modifier.

## public Access Modifier

The Java access modifier public means that all code can access the class, field, constructor or method, regardless of where the accessing code is located. The accessing code can be in a different class and different package.

Here is a public access modifier example:

**public** class Clock {

**public** long time = 0;

}

public class ClockReader {

Clock clock = new Clock();

public long readClock{

return clock.time;

}

}

The time field in the Clock class is marked with the public Java access modifier. Therefore, the ClockReaderclass can access the time field in the Clock no matter what package the ClockReader is located in.

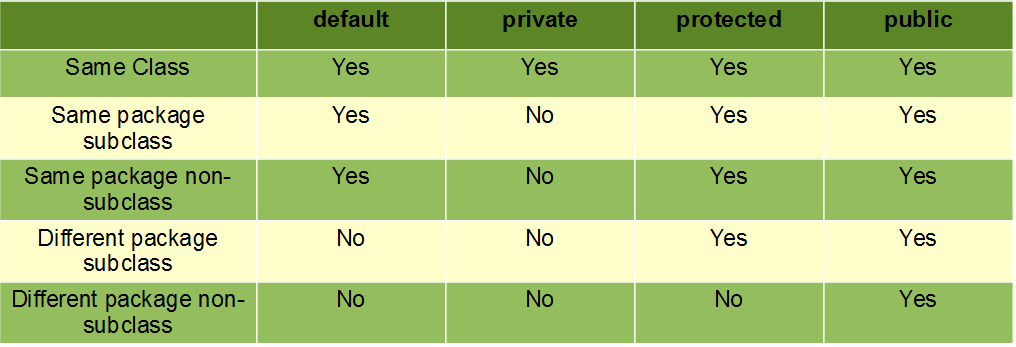
## Class Access Modifiers

It is important to keep in mind that the Java access modifier assigned to a Java class takes precedence over any access modifiers assigned to fields, constructors and methods of that class. If the class is marked with the default access modifier, then no other class outside the same Java package can access that class, including its constructors, fields and methods. It doesn't help that you declare these fields public, or even public static.

The Java access modifiers private and protected cannot be assigned to a class. Only to constructors, methods and fields inside classes. Classes can only have the default (package) and public access modifier assigned to them.

## **Interface Access Modifiers**

Java interfaces are meant to specify fields and methods that are publicly available in classes that implement the interfaces. Therefore you cannot use the private and protected access modifiers in interfaces. Fields and methods in interfaces are implicitly declared public if you leave out an access modifier, so you cannot use the default access modifier either (no access modifier).



**Root class:-**

The **java.lang.Object** class is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.

-**Object** class is present in **java.lang** package. Every class in Java is directly or indirectly derived from the **Object** class. If a Class does not extend any other class then it is direct child class of **Object** and if extends other class then it is an indirectly derived. Therefore the Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

Declaration:- public class Object

**Constructor:-   
Object()**

This is the Single Constructor.

|  |  |
| --- | --- |
| **Methods** | |
| **Modifier and Type** | **Method and Description** |
| protected [**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) | [**clone**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#clone())()  Creates and returns a copy of this object. |
| boolean | [**equals**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#equals(java.lang.Object))([**Object**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html) obj)  Indicates whether some other object is "equal to" this one. |
| protected void | [**finalize**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#finalize())()  Called by the garbage collector on an object when garbage collection determines that there  are no more references to the object. |
| [**Class**](https://docs.oracle.com/javase/7/docs/api/java/lang/Class.html)<?> | [**getClass**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#getClass())()  Returns the runtime class of this Object. |
| int | [**hashCode**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode())()  Returns a hash code value for the object. |
| void | [**notify**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify())()  Wakes up a single thread that is waiting on this object's monitor. |
| void | [**notifyAll**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notifyAll())()  Wakes up all threads that are waiting on this object's monitor. |
| [**String**](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html) | [**toString**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString())()  Returns a string representation of the object. |
| void | [**wait**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait())()  Causes the current thread to wait until another thread invokes the [**notify()**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify()) method  or the **[notifyAll()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "notifyAll())** method for this object. |
| void | [**wait**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait(long))(long timeout)  Causes the current thread to wait until either another thread invokes the [**notify()**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify()) method or  the **[notifyAll()](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html" \l "notifyAll())** method for this object, or a specified amount of time has elapsed. |
| void | [**wait**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#wait(long,%20int))(long timeout, int nanos)  Causes the current thread to wait until another thread invokes the [**notify()**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notify()) method or the  [**notifyAll()**](https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#notifyAll()) method for this object, or some other thread interrupts the current thread, or a  certain amount of real time has elapsed. |

**Some of common type of class in java are as following:-**

1. Wrapper Class:-

2. Mutable Class:-

3. [Abstract Class:-](http://tutorialspointexamples.com/abstract-class-in-java/)

4. [Final Class:-](http://tutorialspointexamples.com/final-keyword-in-java/)

5. [Anonymous Class:-](http://tutorialspointexamples.com/instance-initializer-block-in-java/)

6.[Input-Output Class:-](http://tutorialspointexamples.com/java-input-output-stream-examples-programs-tutorial/)

7. [String Class:-](http://tutorialspointexamples.com/java-string-handling-programs-examples-tutorial-output/)

8. System Class:-

9. Network Class

# Wrapper Classes in Java

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

**Need of Wrapper Classes**

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**

[](http://cdncontribute.geeksforgeeks.org/wp-content/uploads/Wrapper-Class.png)

**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.  
Example:

|  |
| --- |
| // Java program to demonstrate Autoboxing    import java.util.ArrayList;  class Autoboxing  {      public static void main(String[] args)      {          char ch = 'a';            // Autoboxing- primitive to Character object conversion          Character a = ch;            ArrayList<Integer> arrayList = new ArrayList<Integer>();            // Autoboxing because ArrayList stores only objects          arrayList.add(25);            // printing the values from object          System.out.println(arrayList.get(0));      }  } |

Run on IDE

Output:

25

**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.

|  |
| --- |
| // Java program to demonstrate Unboxing  import java.util.ArrayList;    class Unboxing  {      public static void main(String[] args)      {          Character ch = 'a';            // unboxing - Character object to primitive conversion          char a = ch;            ArrayList<Integer> arrayList = new ArrayList<Integer>();          arrayList.add(24);            // unboxing because get method returns an Integer object          int num = arrayList.get(0);            // printing the values from primitive data types          System.out.println(num);      }  } |

Run on IDE

Output:

24

**Implementation**

|  |
| --- |
| // Java program to demonstrate Wrapping and UnWrapping  // in Java Classes  class WrappingUnwrapping  {      public static void main(String args[])      {          //  byte data type          byte a = 1;            // wrapping around Byte object          Byte byteobj = new Byte(a);            // int data type          int b = 10;            //wrapping around Integer object          Integer intobj = new Integer(b);            // float data type          float c = 18.6f;            // wrapping around Float object          Float floatobj = new Float(c);            // double data type          double d = 250.5;            // Wrapping around Double object          Double doubleobj = new Double(d);            // char data type          char e='a';            // wrapping around Character object          Character charobj=e;            //  printing the values from objects          System.out.println("Values of Wrapper objects (printing as objects)");          System.out.println("Byte object byteobj:  " + byteobj);          System.out.println("Integer object intobj:  " + intobj);          System.out.println("Float object floatobj:  " + floatobj);          System.out.println("Double object doubleobj:  " + doubleobj);          System.out.println("Character object charobj:  " + charobj);            // objects to data types (retrieving data types from objects)          // unwrapping objects to primitive data types          byte bv = byteobj;          int iv = intobj;          float fv = floatobj;          double dv = doubleobj;          char cv = charobj;            // printing the values from data types          System.out.println("Unwrapped values (printing as data types)");          System.out.println("byte value, bv: " + bv);          System.out.println("int value, iv: " + iv);          System.out.println("float value, fv: " + fv);          System.out.println("double value, dv: " + dv);          System.out.println("char value, cv: " + cv);      }  } |

Run on IDE

**Output:**

Values of Wrapper objects (printing as objects)

Byte object byteobj: 1

Integer object intobj: 10

Float object floatobj: 18.6

Double object doubleobj: 250.5

Character object charobj: a

Unwrapped values (printing as data types)

byte value, bv: 1

int value, iv: 10

float value, fv: 18.6

double value, dv: 250.5

char value, cv: a

**Mutable Class:-**

1. **Mutable object** – You can change the states and fields after the object is created. For examples: StringBuilder, java.util.Date and etc.

2. **Immutable object** – You cannot change anything after the object is created. For examples: String, boxed primitive objects like Integer, Long and etc.

To create immutable class in java, you have to do following steps.

1. Declare the class as final so it can’t be extended.
2. Make all fields private so that direct access is not allowed.
3. Don’t provide setter methods for variables
4. Make all **mutable fields final** so that it’s value can be assigned only once.
5. Initialize all the fields via a constructor performing deep copy.
6. Perform cloning of objects in the getter methods to return a copy rather than returning the actual object reference

## Abstract class in real world:

Let us take an example of graphic objects. Different graphic objects are there such as circle, rectangle, triangle etc. They all have state defined by their positions, colour etc. and behaviour defined by draw, resize, calculate size etc. As all these object types has common things but with different implementations. We can take advantage of common things with different implementations and put these common things in an abstract class (say GraphicObjects) then extends this class in subclasses to provide specific implementations.

## Abstract class in java:

Abstract class is a way of implementing 0 to 100% abstraction. A class declared with abstract keyword is known as an abstract class. An abstract class may or may not contain abstract method. Abstract classes cannot be instantiated.

## Syntax:

|  |
| --- |
| **abstract class className {**  **// declare fields**  **// declare abstract/non-abstract methods**  **}** |

## Abstract method:

A method with no implementation i.e. without braces and followed by a semicolon.

## Syntax:

|  |
| --- |
| **abstract return\_type methodName();** |

## Example:

|  |
| --- |
| ***/\*\****  ***\* This program is used to show simple use of abstract class.***  ***\* @author CodesJava***  ***\*/***  **abstract** **class** GraphicObjects{  *//abstract method declaration*  **abstract** **void** showShape();  }    **class** Circle **extends** GraphicObjects{  ***/\*\****  ***\* This is the overridden method, provide implementation***  ***\* of abstract method according to your need.***  ***\* @author CodesJava***  ***\*/***  **void** showShape() {  System.out.println("Object type is Circle.");  }  }    **class** Rectangle **extends** GraphicObjects{  ***/\*\****  ***\* This is the overridden method, provide implementation***  ***\* of abstract method according to your need.***  ***\* @author CodesJava***  ***\*/***  **void** showShape() {  System.out.println("Object type is Rectangle.");  }  }    **class** Triangle **extends** GraphicObjects{  ***/\*\****  ***\* This is the overridden method, provide implementation***  ***\* of abstract method according to your need.***  ***\* @author CodesJava***  ***\*/***  **void** showShape() {  System.out.println("Object type is Triangle.");  }    }    **public** **class** AbstractClassExample1 {  **public** **static** **void** main(String args[]){  *//GraphicObjects is the super class*  *//hence it's reference can contain subclass object.*  GraphicObjects obj = **new** Circle();  obj.showShape();  obj = **new** Rectangle();  obj.showShape();  obj = **new** Triangle();  obj.showShape();  }  } |

## Output:

|  |
| --- |
| Object type is Circle.  Object type is Rectangle.  Object type is Triangle. |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/AbstractClassExample1.rar)

## If a class have one abstract method it must be an abstract class but vice versa is not true i.e. it is not necessary that an abstract class have an abstract method.

## Example:

|  |
| --- |
| **abstract** **class** GraphicObjects{*//no error*  *//non-abstract method declaration*  **void** showShape(){  System.out.println("Print object shape.");  }  } |
| *//error here class must be abstract if it*  *//have one or more abstract methods.*  **class** GraphicObjects{  *//abstract method declaration*  **abstract** **void** showShape(){  } |

## If a class extends abstract class than either it has to provide implementation of all abstract methods or declare this class as abstract class.

## Example:

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that a class either***  ***\* have to provide implementation***  ***\* of all abstract methods of extended abstract***  ***\* class or declare abstract itself.***  ***\* @author CodesJava***  ***\*/***  **abstract** **class** GraphicObjects{  *//abstract method declaration*  **abstract** **void** showShape();  }    **class** Circle **extends** GraphicObjects{  ***/\*\****  ***\* This is the overridden method, provide implementation***  ***\* of abstract method according to your need.***  ***\* @author CodesJava***  ***\*/***  **void** showShape() {  System.out.println("Object type is Circle.");  }  }    **class** Rectangle **extends** GraphicObjects{  *//error here, Rectangle class have to provide implementation*  *//of all abstract methods of extended abstract class.*  }    **abstract** **class** Triangle **extends** GraphicObjects{  *//no error here, because Triangle class is declared*  *//as an abstract class*  }    **public** **class** AbstractClassExample2 {  **public** **static** **void** main(String args[]){  *//GraphicObjects is the super class*  *//hence it's reference can contain subclass object.*  GraphicObjects obj = **new** Circle();  obj.showShape();  }  } |

## Output:

|  |
| --- |
| Error. |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/AbstractClassExample2.rar)

## An abstract class can have both static and non-static data members and methods like any other java class.

## Example:

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that abstract class can have both static***  ***\* and non-static data members and methods like any other java class.***  ***\* @author CodesJava***  ***\*/***  **abstract** **class** GraphicObjects{  *//non static data member*  **int** var1 = 50;    *//static data member*  **static** String str1 = "www.codesjava.com";    *//abstract method declaration*  **abstract** **void** showShape();    *//non abstract, non static method*  **void** area(**int** area){  System.out.println("Area = " + area);  }    *//non abstract, static method*  **static** **void** displayGraphicObjects(){  System.out.println("Graphic objects.");  }  }    **class** Circle **extends** GraphicObjects{  ***/\*\****  ***\* This is the overridden method, provide implementation***  ***\* of abstract method according to your need.***  ***\* @author CodesJava***  ***\*/***  **void** showShape() {  System.out.println("Object type is Circle.");  System.out.println("Non static variable = " + var1);  }  }    **public** **class** AbstractClassExample3 {  **public** **static** **void** main(String args[]){  *//GraphicObjects is the super class*  *//hence it's reference can contain subclass object.*  GraphicObjects obj = **new** Circle();  obj.showShape();  obj.area(250);    *//call static method and variable with class name.*  GraphicObjects.displayGraphicObjects();  System.out.println("static variable = "  + GraphicObjects.str1);  }  } |

## Output:

|  |
| --- |
| Area = 250  Graphic objects.  **static** variable = www.codesjava.com |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/AbstractClassExample3.rar)

## Why abstract class is used:

Abstract class in java is used to implement 0 to 100% abstraction.

##### Note: Abstract class provide 0 to 100% abstraction because it may contain no abstract method or it may contain some of its methods as abstract methods or it may contain all its methods as abstract methods.

## Can abstract class have constructors in Java?

Yes, abstract class have constructors in java. But it is not used to instantiate abstract class. It is used in constructor chaining or to initialize abstract class common variables.

## Can abstract class be final in Java?

No, abstract class can’t be final in Java because abstract classes are used only by extending and if they made final they can’t extended.

final is a keyword in java which can be used with instance variables, local variables , methods and classes.

## Use of final keyword in java:

## 1. final variable in java:

A variable declared with final keyword is known as final variable. It may be member variable or local variable. final variables are constants in java and they are generally declared with static keyword. As final variables are treated as constants they can’t reassign. They are initialised at the time of declaration.

## Example:

**FinalExample1.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that the value of***  ***\* final variable can't be change.***  ***\* @author CodesJava***  ***\*/***  **class** Test{  *//final variable*  **final** **int** num = 100;    *//method for try to change the value of final variable.*  **public** **void** show(){  *//error because value of final variable can't be change.*  num = 200;  System.out.println("Num = " + num);  }  }  **public** **class** FinalExample1 {  **public** **static** **void** main(String args[]){  *//creating object of Test Class*  Test obj = **new** Test();  *//method call*  obj.show();  }  } |

## Output:

|  |
| --- |
| Exception in thread "main" java.lang.Error:  Unresolved compilation problem:  The **final** field Test.num cannot be assigned  at com.codesjava.business.Test.show(FinalExample1.java:15)  at com.codesjava.business.FinalExample1.main  (FinalExample1.java:24) |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample1.rar)

##### Note: Inside Anonymous classes only final variables are accessible.

## 2.  final method in java:

A method declared with final keyword is known as final method.

## Example:

**FinalExample2.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that final method can't be override.***  ***\* @author CodesJava***  ***\*/***  **class** Show{  **public** **final** **void** show(){  System.out.println("Hello world.");  }  }    **class** Display **extends** Show{  *//error because final method can't be override.*  **public** **void** show(){  System.out.println("Hello codesjava.com.");  }  }    **public** **class** FinalExample2 {  **public** **static** **void** main(String args[]){  *//creating object of Display class*  Display obj = **new** Display();  *//method call*  obj.show();  }  } |

## Output:

|  |
| --- |
| Exception in thread "main" java.lang.VerifyError:  **class** com.codesjava.business.Display overrides **final** method show.()V  at java.lang.ClassLoader.defineClass1(**Native** Method)  at java.lang.ClassLoader.defineClass(Unknown Source)  at java.security.SecureClassLoader.defineClass(Unknown Source)  at java.net.URLClassLoader.defineClass(Unknown Source)  at java.net.URLClassLoader.access$100(Unknown Source)  at java.net.URLClassLoader$1.run(Unknown Source)  at java.net.URLClassLoader$1.run(Unknown Source)  at java.security.AccessController.doPrivileged(**Native** Method)  at java.net.URLClassLoader.findClass(Unknown Source)  at java.lang.ClassLoader.loadClass(Unknown Source)  at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)  at java.lang.ClassLoader.loadClass(Unknown Source)  at com.codesjava.business.FinalExample2.main  (FinalExample2.java:23) |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample2.rar)

##### Note: A final method can be inherited but can’t be override.

## 3.  final class in java:

A class declared with final keyword is known as final class. A final class can’t be inherited.

## Example:

**FinalExample3.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that final class can't be inherited.***  ***\* @author CodesJava***  ***\*/***  **final** **class** Show{  **public** **void** show(){  System.out.println("Hello world.");  }  }  *//error because final class can't be inherited.*  **class** Display **extends** Show{  **public** **void** display(){  System.out.println("Hello codesjava.com.");  }  }    **public** **class** FinalExample3 {  **public** **static** **void** main(String args[]){  *//creating object of Display class*  Display obj = **new** Display();  *//method call*  obj.display();  }  } |

## Output:

|  |
| --- |
| Exception in thread "main" java.lang.Error:  Unresolved compilation problem:  The type Display cannot subclass the **final** **class** Show  at com.codesjava.business.Display.(FinalExample3.java:13)  at com.codesjava.business.FinalExample3.main  (FinalExample3.java:22) |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample3.rar)

##### Note: final method can be inherited.

## Example:

**FinalExample4.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that final method can be inherited.***  ***\* @author CodesJava***  ***\*/***  **class** Show{  **public** **final** **void** show(){  System.out.println("Hello world.");  }  }    **class** Display **extends** Show{  **public** **void** display(){  System.out.println("Hello codesjava.com.");  }  }    **public** **class** FinalExample4 {  **public** **static** **void** main(String args[]){  *//creating object of Display class*  Display obj = **new** Display();  *//method call*  obj.show();  }  } |

## Output:

|  |
| --- |
| Hello world. |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample4.rar)

##### Note: Abstract class and constructor can’t be final.

## 4. Blank final variable in java:

A variable declared with final keyword but not initialised at declaration time is known as blank final variable. They are initialised at the time of object creation in constructor and can’t change after that.

## Example:

**FinalExample5.java**

|  |
| --- |
| ***/\*\****  ***\* This class is used to show the example of blank final variable.***  ***\* @author codesjava***  ***\*/***  **class** Test{  *//blank final variable which can only be*  *//initialize through constructor.*  **final** **int** num;    Test(**int** n){  num = n;  System.out.println("Num = " + num);  }  }  **public** **class** FinalExample5 {  **public** **static** **void** main(String args[]){  **new** Test(100);  }  } |

## Output:

|  |
| --- |
| Num = 100 |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample5.rar)

## 5. static blank final variable in java:

A static variable declared with final keyword but not initialised at declaration time is known as static blank final variable. It can be initialised in a static block only.

## Example:

**FinalExample6.java**

|  |
| --- |
| ***/\*\****  ***\* This class is used to show the example of static blank final variable.***  ***\* @author CodesJava***  ***\*/***  **class** Test{  *//blank final variable which can only be initialize*  *//through static initializer block.*  **static** **final** **int** num;    **static**{  num = 100;  System.out.println("Num = " + num);  }  }  **public** **class** FinalExample6 {  **public** **static** **void** main(String args[]){  **new** Test();  }  } |

## Output:

|  |
| --- |
| Num = 100 |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample6.rar)

## 6.  final parameter in java:

A method parameter declared with final keyword is known as final parameter. Its value can’t be changed.

## Example:

**FinalExample7.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that value of***  ***\* final parameter can't be changed.***  ***\* @author CodesJava***  ***\*/***  **class** Test{  **public** **void** showDouble(**final** **int** num){  *//error because value of final parameter can't be changed.*  num = num \* 2;  System.out.println("Num \* 2 = " + num);  }  }  **public** **class** FinalExample7 {  **public** **static** **void** main(String args[]){  *//creating object of Test class*  Test obj = **new** Test();  *//method call*  obj.showDouble(10);  }  } |

## Output:

|  |
| --- |
| Exception in thread "main" java.lang.Error:  Unresolved compilation problem:  The **final** local variable num cannot be assigned.  It must be blank and not using a compound assignment  at com.codesjava.business.Test.showDouble  (FinalExample7.java:11)  at com.codesjava.business.FinalExample7.main  (FinalExample7.java:20) |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/FinalExample7.rar)

## Advantages/Benefits of final keyword:

1. **Performance: JVM kept in cache if variables, methods and classes are declared final.**
2. **Immutable classes: With the help of final keyword we can made immutable classes.**

**Anonymous Class**

## Instance initializer block:

Instance initializer block is a mechanism provided by java compiler to define a group of statements common to all constructors at a single place. At the compilation time, compiler moves these statements at the beginning of all constructors after super. It is can also be used to initialize the instance variable.

## Example:

**AnonymousBlockExample1.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show the use of AnonymousBlock.***  ***\* @author CodesJava***  ***\*/***  **class** Display {  **int** a, b;    *//Anonymous or instance initializer Block*  {  System.out.println("AnonumousBlock called.");  a = 10;  }    *//default constructor*  Display(){  System.out.println("default constructor called.");  }    *//one argument constructor*  Display(**int** num){  System.out.println("one parameter constructor called.");  b = num;  }    *//method to display values*  **public** **void** display(){  System.out.println("a = " + a);  System.out.println("b = " + b);  }  }    **public** **class** AnonymousBlockExample1 {  **public** **static** **void** main(String args[]){  Display obj1 = **new** Display();  obj1.display();    Display obj2 = **new** Display(20);  obj2.display();  }  } |

## Output:

|  |
| --- |
| AnonumousBlock called.  **default** constructor called.  a = 10  b = 0  AnonumousBlock called.  one parameter constructor called.  a = 10  b = 20 |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/AnonumousBlockExample1.rar)

## If two Anonymous Blocks are used then they will execute in the same order in which they are appear.

## Example:

**AnonymousBlockExample2.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that if two AnonymousBlocks***  ***\* are used then they will execute in the same order in***  ***\* which they are appear.***  ***\* @author CodesJava***  ***\*/***  **class** Display {  **int** a, b, c;    *//First Anonymous or instance initializer Block*  {  System.out.println("First AnonumousBlock called.");  a = 10;  }    *//Second Anonymous or instance initializer Block*  {  System.out.println("Second AnonumousBlock called.");  b = 20;  }    *//default constructor*  Display(){  System.out.println("default constructor called.");  }    *//one argument constructor*  Display(**int** num){  System.out.println("one parameter constructor called.");  c = num;  }    *//method to display values*  **public** **void** display(){  System.out.println("a = " + a);  System.out.println("b = " + b);  System.out.println("c = " + c);  }  }    **public** **class** AnonymousBlockExample2 {  **public** **static** **void** main(String args[]){  Display obj1 = **new** Display();  obj1.display();    Display obj2 = **new** Display(30);  obj2.display();  }  } |

## Output:

|  |
| --- |
| First AnonumousBlock called.  Second AnonumousBlock called.  **default** constructor called.  a = 10  b = 20  c = 0  First AnonumousBlock called.  Second AnonumousBlock called.  one parameter constructor called.  a = 10  b = 20  c = 30 |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/AnonumousBlockExample2.rar)

## If static and non-static Anonymous Blocks are used then static Anonymous Block is executed only once.

## Example:

**AnonymousBlockExample3.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that if static and non-static***  ***\* AnonymousBlocks are used then static AnonymousBlocks is***  ***\* executed only once.***  ***\* @author CodesJava***  ***\*/***  **class** Display {  **int** a, b;    *//static Anonymous or instance initializer Block*  **static** {  System.out.println("Static AnonumousBlock called.");  }    *//non-static Anonymous or instance initializer Block*  {  System.out.println("Non-Static AnonumousBlock called.");  a = 20;  }    *//default constructor*  Display(){  System.out.println("default constructor called.");  }    *//one argument constructor*  Display(**int** num){  System.out.println("one parameter constructor called.");  b = num;  }    *//method to display values*  **public** **void** display(){  System.out.println("a = " + a);  System.out.println("b = " + b);  }  }    **public** **class** AnonymousBlockExample3 {  **public** **static** **void** main(String args[]){  Display obj1 = **new** Display();  obj1.display();    Display obj2 = **new** Display(30);  obj2.display();  }  } |

## Output:

|  |
| --- |
| **Static** AnonumousBlock called.  Non-**Static** AnonumousBlock called.  **default** constructor called.  a = 20  b = 0  Non-**Static** AnonumousBlock called.  one parameter constructor called.  a = 20  b = 30 |

[***Download this example.***](https://codesjava.com/wp-content/uploads/2014/08/AnonumousBlockExample3.rar)

## In which order static initializer  block, instance initialize  block, super and constructor are called?

**static initialize  block – super- instance initialize  block – constructor.**

## Example:

**AnonymousBlockExample4.java**

|  |
| --- |
| ***/\*\****  ***\* This program is used to show that in which order static***  ***\* AnonumousBlocks, non-static AnonumousBlocks, super and***  ***\* default constructors are called.***  ***\* @author CodesJava***  ***\*/***  **class** Show{  Show(){  System.out.println("Super class constructor.");  }  }    **class** Display **extends** Show{  *//static Anonymous or instance initializer Block*  **static** {  System.out.println("Static AnonumousBlock called.");  }    *//non-static Anonymous or instance initializer Block*  {  System.out.println("Non-Static AnonumousBlock called.");  }    *//default constructor*  Display(){  **super**();  System.out.println("default constructor called.");  }  }    **public** **class** AnonymousBlockExample4 {  **public** **static** **void** main(String args[]){  Display obj = **new** Display();  }  } |

## Output:

|  |
| --- |
| **Static** AnonumousBlock called.  **Super** **class** constructor.  Non-**Static** AnonumousBlock called.  **default** constructor called. |

# Java.lang.Class class in Java