Interfaces

For any big software project disparate groups of programmers need to work together. The groups agree to a "contract" that spells out how their software interacts. Each group should be able to write their code without any knowledge of how the other group's code is written. Generally speaking, *interfaces* are such contracts.

For example, imagine a futuristic society where computer-controlled robotic cars transport passengers through city streets without a human operator. Automobile manufacturers write software that operates the automobile—stop, start, accelerate, turn left, and so forth. Another industrial group, electronic guidance instrument manufacturers, makes computer systems that receive GPS (Global Positioning System) position data and wireless transmission of traffic conditions and use that information to drive the car.

The auto manufacturers must publish an industry-standard interface that spells out in detail what methods can be invoked to make the car move (any car, from any manufacturer). The guidance manufacturers can then write software that invokes the methods described in the interface to command the car. Neither industrial group needs to know *how* the other group's software is implemented. In fact, each group considers its software highly proprietary and reserves the right to modify it at any time, as long as it continues to adhere to the published interface.

Interfaces in Java

Example of nested type shown below

An *interface* is a reference type, similar to a class and have following characteristics:

- Contains *only* constants, method signatures, default methods, static methods, and **nested types**.
- Method bodies exist only for default methods and static methods.
- Interfaces cannot be instantiated—they can only be *implemented* by classes or *extended* by other interfaces.

The purpose of Interface is to just declare contract. Your client will implement the methods and for that it must be **public** by default.

Method signatures have no braces and are terminated with a semicolon.

```
public interface OperateCar {

  // constant declarations, if any
  // method signatures
  // An enum with values RIGHT, LEFT

  int turn(Direction direction,  double radius, double startSpeed,  double endSpeed);
  int changeLanes(Direction direction, double startSpeed,  double endSpeed);
  int signalTurn(Direction direction, boolean signalOn);
  int getRadarFront(double distanceToCar, double speedOfCar);
  int getRadarRear(double distanceToCar,  double speedOfCar);
      ......
}
```

An example containing nested types along with other elements of interface.

```
public interface Input
{
   public static class KeyEvent {
       public static final int KEY_DOWN = 0;        public static final int KEY_UP = 1;
       public int type;                             public int keyCode;
       public char keyChar;
   }
   public static class TouchEvent {
       public static final int TOUCH_DOWN = 0;        public static final int TOUCH_UP = 1;
       public static final int TOUCH_DRAGGED = 2;
       public int type;        public int x, y;        public int pointer;
   }
   public boolean isKeyPressed(int keyCode);
   public boolean isTouchDown(int pointer);
   public int getTouchX(int pointer);
   public int getTouchY(int pointer);
   public float getAccelX();      public float getAccelY();      public float getAccelZ();
   public List<KeyEvent> getKeyEvents();
   public List<TouchEvent> getTouchEvents();
}
```

Interfaces as APIs

An *Application Programming Interface (API)* defines the interfaces by which one piece of software communicates with another at the source level. It provides abstraction by providing a standard set of interfaces - usually functions - that one piece of software (typically a higher-level piece) can invoke from another piece of software (usually a lower-level piece). APIs are also common in commercial software products. Typically, a company sells a software package that contains complex methods that another company wants to use in its own software product.

For example, an API might abstract the concept of drawing text on the screen through a family of functions that provide everything needed to draw the text. The API merely defines the interface; the piece of software that actually provides the API is known as the implementation of the API.

An example would be a package of digital image processing methods that are sold to companies making end-user graphics programs. The image processing company writes its classes to implement an interface, which it makes public to its customers. The graphics company then invokes the image processing methods using the signatures and return types defined in the interface. While the image processing company's API is made public (to its customers), its implementation of the API is kept as a closely guarded secret—in fact, it may revise the implementation at a later date as long as it continues to implement the original interface that its customers have relied on.

Defining an Interface

An interface declaration consists of modifiers, the keyword interface, the interface name, a comma-separated list of parent interfaces (if any), and the interface body. For example:

```
public interface GroupedInterface extends Interface1, Interface2, Interface3 {

   // constant declarations

   // base of natural logarithms
   double E = 2.718282;

   // method signatures
   void doSomething (int i, double x);
   int doSomethingElse(String s);
}
```

- The public access specifier indicates that the interface can be used by any class in **any package**.
- If not public then your interface is accessible only to classes defined in the **same** package as the interface.
- An interface can extend other interfaces.
- A class can extend only one other class in Java, but an interface can extend any number of interfaces.

The Interface Body

The interface body can contain abstract methods, default methods, and static methods.

- An **abstract** method within an interface is followed by a semicolon, but no braces (an abstract method does not contain an implementation).
- **Default** methods are defined with the default modifier, and
- **Static** methods with the static keyword. All abstract, default, and static methods in an interface are implicitly public, so you can omit the public modifier.
- In addition, an interface can contain constant declarations (fields which acts as constants : it can't be instantiated without implementation and it must not be inherited also). So all constant values defined in an interface are **implicitly public, static, and final**. Once again, you can omit these modifiers.

Examples :

- **One Interface - One Class** : Define a java shape interface and implemented it using the circle class . Both shape interface and circle class are defined within a Main.java file. As you may notice Main class is the only public class in this file.

```
interface shape
{
    public String baseclass="shape";
    public void Draw();
}
```

```
class circle implements shape
 {
   public void Draw() {   System.out.println("Drawing Circle”);
   }
 }
```

```
public class Main {
    public static void main(String[] args) {
        shape circleshape=new circle();          circleshape.Draw();
    }
}
```

Shape interface has only one constant and one abstract method with no body.

- **Interface Inherits An Interface :** An interface can inherit one or more interfaces using **extends** keyword, but no interface can implements any other interfaces.

```
public class Main {
   public static void main(String[] args) {
        shapeA circleshape=new circle();
        circleshape.Draw();
        circleshape.Draw2();
   }
}
```

```
interface shapeA
{
   public  String baseclass="shape";
   public void Draw();
}
interface shapeB extends shapeA
{
   public  String baseclass="shape2";
   public void Draw2();
}
```

Please note multi-declaration of member variable ***baseclass***, even that all interface fields are **finals** we can redefine the same while extending or/ implementing an interface.

```
class circle implements shapeB
{
   public  String baseclass="shape3";
   public void Draw() {
      System.out.println("Drawing Circle here:"+baseclass);
   }
   @Override
   public void Draw2() {
       System.out.println("Drawing Circle here:"+baseclass);
   }
 }
```

Note : @Override annotation informs the compiler that the element is meant to override an element declared in a supertype (i.e. interface).

The following Sports interface is extended by Hockey and Football interfaces.

```
//Filename: Sports.java
public interface Sports
{
  public void setHomeTeam(String name);
  public void setVisitingTeam(String name);
}
```

```
//Filename: Football.java
public interface Football extends Sports
{
  public void homeTeamScored(int points);
  public void visitingTeamScored(int points);
  public void endOfQuarter(int quarter);
}
```

```
//Filename: Hockey.java
public interface Hockey extends Sports
{
  public void homeGoalScored();
  public void visitingGoalScored();
  public void endOfPeriod(int period);
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

```
  public void overtimePeriod(int ot);
}
```

Using an Interface as a Type

If you define a reference variable whose type is an interface, any object you assign to it *must* be an instance of a class that implements the interface.

Examples :

Ex:1
```
public class MyClass extends ParentClass implements Interface1, Interface2 {
    //some code
    }
```

This class can be used in different places as follows:

```
MyClass m1 = new MyClass();          ParentClass p = new MyClass();
Interface1 i1 = new MyClass();       Interface2 i2 = new MyClass();
```

If variable is declared to be the type of an interface, its value can reference any object that is instantiated from any class that implements the interface.

Evolving Interfaces

Consider an interface that you have developed called DoIt:

```
public interface DoIt {
  void doSomething(int i, double x);      int doSomethingElse(String s);
}
```

Suppose that, at a later time, you want to add a third method to DoIt, so that the interface now becomes:

```
public interface DoIt {
  void doSomething(int i, double x);      int doSomethingElse(String s);
  boolean didItWork(int i, double x, String s);
}
```

If you make this change, then all classes that implement the old DoIt interface will break because they no longer implement the old interface. Programmers relying on this interface will protest loudly.

If you want to add additional methods to an interface, you have several options.

1. You could create a DoItPlus interface that extends DoIt:

```
public interface DoItPlus extends DoIt {
  boolean didItWork(int i, double x, String s);
}
```

Now users of your code can choose to continue to use the old interface or to upgrade to the new interface.

2. Alternatively, you can define your new methods as **default methods**. The following example defines a default method named didItWork:

```
public interface DoIt {
  void doSomething(int i, double x);
  int doSomethingElse(String s);
  default boolean didItWork(int i, double x, String s) {
     // Method body
  }
}
```

Note that you must provide an implementation for default methods. You could also define new static methods to existing interfaces. Users who have classes that implement interfaces enhanced with new default or static methods do not have to modify or recompile them to accommodate the additional methods.

Default Methods (Java 8)

If we add some new methods to already developed interfaces, then programmers who have implemented those interfaces would have to rewrite their implementations. If they add them as **static methods**, then programmers would regard them as utility methods, not as essential, core methods.

To overcome this limitation of extending interface a new concept is introduced in Java 8 called default methods which is also referred to as Defender Methods or Virtual extension methods. **Default** methods are those methods which have some default implementation and helps in evolving the interfaces without breaking the existing code.

```
public interface SimpleInterface {
     public void doSomeWork();
    //A default method in the interface created using "default" keyword
     default public void doSomeOtherWork(){
       System.out.println("DoSomeOtherWork implementation in the interface");
     }
}
class SimpleInterfaceImpl implements SimpleInterface{
     @Override
     public void doSomeWork() {
       System.out.println("Do Some Work implementation in the class");
     }
     //   Not required to override to provide an implementation for doSomeOtherWork.

     public static void main(String[] args) {
       SimpleInterfaceImpl simpObj = new SimpleInterfaceImpl();
       simpObj.doSomeWork();
       simpObj.doSomeOtherWork();
     }
   }
```

Static Methods

- A static method is a method that is associated with the class in which it is defined rather than with any object. Every instance of the class shares its static methods.

This makes it easier for you to organize helper methods in your libraries; you can keep static methods specific to an interface in the same interface rather than in a separate class. The following example defines a static method that retrieves a **ZoneId** object corresponding to a time zone identifier; it uses the system default time zone if there is no ZoneId object corresponding to the given identifier. (As a result, you can simplify the method getZonedDateTime):

```
public interface TimeClient {
  // ...
  static public ZoneId getZoneId (String zoneString) {
    try {
      return ZoneId.of(zoneString);
    } catch (DateTimeException e) {
      System.err.println("Invalid time zone: " + zoneString +
        "; using default time zone instead.");
      return ZoneId.systemDefault();
    }
  }

  default public ZonedDateTime getZonedDateTime(String zoneString) {
    return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
  }
}
```

All method declarations in an interface, including static methods, are implicitly public, so you can omit the public modifier.

Nested Interface in Java

We can declare interfaces as member of a class or another interface. Such an interface is called as member interface or nested interface.

Nested interface which is declared within the interface	**Nested interface which is declared within the class**

interface Showable{
  void show();
  interface Message{ void msg(); }
}
class TestNestedInterface1 implements
          Showable.Message {
  public void msg(){
    System.out.println("Hello nested interface");
	class A {
	  interface Message{ void msg(); }
	}
	
	class TestNestedInterface2 implements
	          A.Message {
	 public void msg() {
	    System.out.println("Hello nested interface");

}
public static void main(String args[]) {
//upcasting here
Showable.Message message=
new TestNestedInterface1();
message.msg();
}
}
Output: Hello nested interface
	}
	public static void main(String args[]) {
	//upcasting here
	A.Message message=
	new TestNestedInterface2();
	message.msg();
	}
	}
	Output: Hello nested interface

As you can see in the above example, we are accessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first. In collection framwork, sun microsystem has provided a nested interface Entry. Entry is the subinterface of Map i.e. accessed by Map.Entry.

Example of defining class inside an interface :

If functionality of a class is closely associated with an interface and we are not going to use that class anywhere then we can define a class inside an interface.

```
interface Employee {
    class PersonalDetails {
            String empName = "Ashok";
            int age = 25;
            String dob = "15-June-1991";
            void empDetails() {  System.out.println(empName+ " - " + age + " - " +dob);              }
    }
     class Designation {
            void empDesg() {   System.out.println("Software Engineer");     }
    }
}

public class Test {
     public static void main(String args[]) {
            Employee.PersonalDetails empDetails = new Employee.PersonalDetails();
            empDetails.empDetails();
            Employee.Designation empDesignation = new Employee.Designation();
            empDesignation.empDesg();
    }
}
```

Another Example :

```
public interface VehicleService {
  public void repair(Vehicle v);
  public class Vehicle{
    String vehicleModel;
    String vehicleNumber;
    public Vehicle(String vehicleModel, String vehicleNumber) {
        super();
        this.vehicleModel = vehicleModel;
        this.vehicleNumber = vehicleNumber;
    }

  }
}
```

1. In the above case **vehicle** class is available for VehicleService and we are not using it anywhere else.
2. To provide default implementation of an interface we can define a class inside an interface Interface

public interface **VehicleService** {
public void repair();
public class DefaultVehicle implements
VehicleService {
@Override
public void repair() {
System.out.println(" Default Repair");
}
}
	Implementation Class
	
	public class **busRepair** implements VehicleService{
	@Override
	public void repair() {
	System.out.println(" Bus Repair"); }
	public static void main(String args[]) {
	busRepair b = new busRepair();
	b.repair();
	DefaultVehicle d = new DefaultVehicle();
	d.repair();
	}
	}

Abstract Methods and Classes

- An *abstract method* is a method that is declared without an implementation (without braces, and followed by a semicolon), like this: **abstract** void moveTo(double deltaX, double deltaY);
- When a class contains one or more abstract methods, then it should be declared abstract. You must use abstract keyword to make a class abstract.
- We cannot use abstract classes to instantiate objects directly. It needs to be extended and its method needs to be implemented.
- The abstract methods of an abstract class must be defined in its subclass.
- You cannot declare abstract constructors or abstract static methods.
- Abstract class can contain abstract as well as non-abstract methods. It also has a member variables and constructors.
- Private method cannot be abstract.

Example

```
public abstract class GraphicObject {
	// declare fields
	// declare nonabstract methods
	abstract void draw();
}
```

When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, then the subclass must also be declared abstract.

Note: Methods in an *interface* (see the Interfaces section) that are not declared as default or static are *implicitly* abstract, so the abstract modifier is not used with interface methods. (It can be used, but it is unnecessary.)

Abstract Classes vs Interfaces

Abstract classes are similar to interfaces. You cannot instantiate them, and they may contain a mix of methods declared with or without an implementation.

- However, with abstract classes, you can declare fields that are not **static** and **final**, and define public, protected, and private concrete methods.
- With interfaces, all fields are automatically **public, static, and final**, and all methods that you declare or define (as default methods) are public.
- In addition, you can extend only one class, whether or not it is abstract, whereas you can implement any number of interfaces.

Which should you use, abstract classes or interfaces?

- Consider using abstract classes if any of these statements apply to your situation:
    - You want to share code among several closely related classes.
    - You expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
    - You want to declare non-static or non-final fields. This enables you to define methods that can access and modify the state of the object to which they belong.
- Consider using interfaces if any of these statements apply to your situation:
    - You expect that unrelated classes would implement your interface. For example, the interfaces Comparable and Cloneable are implemented by many unrelated classes.
    - You want to specify the behavior of a particular data type, but not concerned about who implements its behavior.
    - You want to take advantage of multiple inheritance of type.

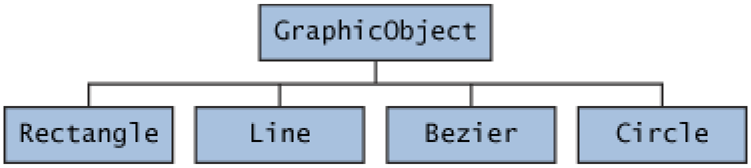
An example of an abstract class in the JDK is AbstractMap, which is part of the Collections Framework. Its subclasses (which include HashMap, TreeMap, and ConcurrentHashMap) share many methods (including get, put, isEmpty, containsKey, and containsValue) that AbstractMap defines.

An example of a class in the JDK that implements several interfaces is HashMap, which implements the interfaces Serializable, Cloneable, and Map<K, V>. By reading this list of interfaces, you can infer that an instance of HashMap (regardless of the developer or company who implemented the class) can be cloned, is serializable (which means that it can be converted into a byte stream; see the section Serializable Objects), and has the functionality of a map. In addition, the Map<K, V> interface has been enhanced with many default methods such as merge and forEach that older classes that have implemented this interface do not have to define.

Note that many software libraries use both abstract classes and interfaces; the HashMap class implements several interfaces and also extends the abstract class AbstractMap.

An Abstract Class Example

In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: moveTo, rotate, resize, draw) in common. Others require different implementations (for example, resize or draw). All GraphicObjects must be able to draw or resize themselves; they just differ in how they do it. This is a perfect situation for an abstract superclass. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object (for example, GraphicObject) as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle Inherit from GraphicObject

First, you declare an abstract class, **GraphicObject**, to provide member variables and methods that are wholly shared by all subclasses. GraphicObject also declares abstract methods for methods, such as draw or resize, that need to be implemented by all subclasses but must be implemented in different ways. The GraphicObject class can look something like this:

```
abstract class GraphicObject {
  int x, y;
  ...
  void moveTo(int newX, int newY) {      ...    }
  abstract void draw();         abstract void resize();
}
```

Each nonabstract subclass of GraphicObject, such as Circle and Rectangle, must provide implementations for the draw and resize methods:

```
class Circle extends GraphicObject {
  void draw() {       ...     }
  void resize() {      ...     }
}
class Rectangle extends GraphicObject {
  void draw() {       ...   }
```

```
   void resize() {        ...    }
}
```

When an Abstract Class Implements an Interface

It was noted that a class that implements an interface must implement *all* of the interface's methods. It is possible, however, to define a class that does not implement all of the interface's methods, provided that the class is declared to be abstract. For example,

```
abstract class X implements Y {
  // implements all but one method of Y
}

class XX extends X {
  // implements the remaining method in Y
}
```

In this case, class X must be abstract because it does not fully implement Y, but class XX does, in fact, implement Y.

Class Members

An abstract class may have static fields and static methods. You can use these static members with a class reference (for example, AbstractClass.staticMethod()) as you would with any other class.

Final Classes and Methods in Java

You can declare some or all of a class's methods *final*. You use the final keyword in a method declaration to indicate that the method cannot be overridden by subclasses. The Object class does this—a number of its methods are final.

You might wish to make a method final if it has an implementation that should not be changed and it is critical to the consistent state of the object. For example, you might want to make the getFirstPlayer method in this ChessAlgorithm class final:

```
class ChessAlgorithm {
   enum ChessPlayer { WHITE, BLACK }
   ...
   final ChessPlayer getFirstPlayer() {       return ChessPlayer.WHITE;     }
   ...
}
```

Methods called from constructors should generally be declared final. If a constructor calls a non-final method, a subclass may redefine that method with surprising or undesirable results.

Note that you can also declare an entire class final. A class that is declared final cannot be subclassed. This is particularly useful, for example, when creating an immutable class like the String class.

Static Keyword

- **Class Variables**

When a number of objects are created from the same class blueprint, they each have their own distinct copies of *instance variables*. Sometimes, you want to have variables that are common to all objects. This is accomplished with the static modifier.

Fields that have the static modifier in their declaration are called ***static fields* or *class variables***. They are associated with the class, rather than with any object. Every instance of the class shares a class variable, which is in one fixed location in memory. Any object can change the value of a class variable, but class variables can also be manipulated without creating an instance of the class.

For example, suppose you want to create a number of Bicycle objects and assign each a serial number, beginning with 1 for the first object. This ID number is unique to each object and is therefore an instance variable. At the same time, you need a field to keep track of how many Bicycle objects have been created so that you know what ID to assign to the next one. Such a field is not related to any individual object, but to the class as a whole. For this you need a class variable, numberOfBicycles, as follows:

```
public class Bicycle {

   private int cadence;    private int gear;    private int speed;

   // add an instance variable for the object ID
   private int id;

   // add a class variable for the number of Bicycle objects instantiated
   private static int numberOfBicycles = 0;
      ...
}
```

In Jave , static means that it's a variable/method of a class, it belongs to the whole class but not to one of its certain objects. This means that static keyword can be used only in a 'class scope'. So we cannot declare a static variable inside a method.

Class variables are referenced by the class name itself, as in Bicycle.numberOfBicycles

Note: You can also refer to static fields with an object reference like myBike.numberOfBicycles but this is **discouraged** because it does not make it clear that they are class variables.

You can use the Bicycle constructor to set the id instance variable and increment the numberOfBicycles class variable:

```
public class Bicycle {

   private int cadence;  private int gear;    private int speed;
   private int id;       private static int numberOfBicycles = 0;

   public Bicycle(int startCadence, int startSpeed, int startGear){
       gear = startGear;    cadence = startCadence;     speed = startSpeed;

       // increment number of Bicycles and assign ID number
       id = ++numberOfBicycles;
   }
   // new method to return the ID instance variable
   public int getID() {  return id;    }
       ...
}
```

- **Class Methods**

The Java programming language also supports static methods. It should be invoked with the class name, without the need for creating an instance of the class, as in ClassName.methodName(args)

Note: You can also refer to static methods with an object reference like instanceName.methodName(args) but this is **discouraged** because it does not make it clear that they are class methods.

A common use for static methods is to access static fields. For example, we could add a static method to the Bicycle class to access the numberOfBicycles static field:

public **static** int getNumberOfBicycles() { return numberOfBicycles; }

Not all combinations of instance and class variables and methods are allowed:

1. Instance methods can access instance variables, instance methods, class variables and class methods directly.
2. Class methods (i.e. with **static** keyword) can access class variables and class methods directly.
3. Class methods ***cannot*** access **instance variables** or **instance methods** directly—they must use an object reference. Also, class methods cannot use the **this** keyword as there is no instance for this to refer to.

- **Constants**

The static modifier, in combination with the final modifier, is also used to define constants. The final modifier indicates that the value of this field cannot change.

For example, the following variable declaration defines a constant named PI, whose value is an approximation of pi (the ratio of the circumference of a circle to its diameter):

static final double PI = 3.141592653589793;

Constants defined in this way cannot be reassigned, and it is a compile-time error if your program tries to do so. By convention, the names of constant values are spelled in uppercase letters. If the name is composed of more than one word, the words are separated by an underscore (_).

Note: If a primitive type or a string is defined as a constant and the value is known at compile time, the compiler replaces the constant name everywhere in the code with its value. This is called a *compile-time constant*. If the value of the constant in the outside world changes (for example, if it is legislated that pi actually should be 3.975), you will need to recompile any classes that use this constant to get the current value.

Static blocks in Java

Java supports a special block, called **static block** (also called static clause) which can be used for static initializations of a class.

- The code inside static block is executed only once: when the **class itself is initialized**, no matter how many objects of that type you create. This means the first time you make an object of that class or the first time you access a static member of that class (even if you never make an object of that class).
- Constructor is invoked while creating an instance of the class. Static block is invoked when a class loader loads this class definition.

For example, check output of following Java program.

```
// filename: Main.java
class Test {
   static int i;
   int j;
   static {     // start of static block
      i = 10;
      System.out.println("static block called ");
   }   // end of static block
   Test(){
      System.out.println("Constructor called");
   }   }
```

```
Output:
static block called
10
```

```
class Main {
   public static void main(String args[]) {

      // Although we don't have an object of Test,
         static block is called because i is being
         accessed in following statement.
         System.out.println(Test.i);
   }
}
```

Also, static blocks are executed before constructors. For example, check output of following Java

```
class Main {
   public static void main(String args[]) {

      // Although we have two objects, static block is executed only once.
      Test t1 = new Test();
      Test t2 = new Test();
   }
}
```

```
Output:
static block called
Constructor called
Constructor called
```

We can also use **multiple static blocks**. They execute in the given order which means the first static block executes before second static block.

Non static blocks in java

It's also possible to have non-static initializer blocks. Those act like extensions to the set of constructor methods defined for the class. They look just like static initializer blocks, except the keyword "static" is left off.

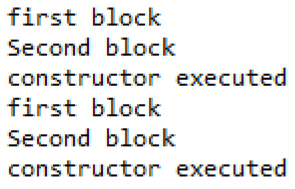
- Whenever object created non static blocks will be executed before the execution of constructor
- Non static blocks are class level block which does not have prototype
- We may need it to execute any logic whenever object is created irrespective of constructor used in object creation.
- Non static blocks are automatically called by JVM for every object creation
- We can create any number of Non static blocks. Order of execution of non-static blocks will be order as they are defined.

```
public class NonStaticBlock {
    {    System.out.println("first block");        }
    {    System.out.println("second block");      }

    NonStsticBlock() {
        System.out.println("constructor called");
    }
}
```

```
Public static void main(String[] args)  {
    NonStaticBlocks  obj = new NonStaticBlocks();
    NonStaticBlocks  obj1 = new NonStaticBlocks();
}
```

Output



```
first block
Second block
constructor executed
first block
Second block
constructor executed
```