Constructors in Java - Constructors are used to **initialize** the **object's state**. Like methods, a constructor also contains collection of statements(i.e. instructions) that are executed at time of Object creation.

- Each time an object is created using new() keyword at least one constructor (it could be default constructor) is invoked to assign initial values to the data members of the same class.
- Constructor(s) of a class must has same name as the class name in which it resides.
- A constructor in Java **can not** be **abstract**, **final**, **static** and **Synchronized**.
- Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.
- Constructors are not **inherited**.

Types of constructor - There are two type of constructor in Java,

1. No-argument constructor: A constructor that has no parameter is known as default constructor.

- If we don't define a constructor in a class, then compiler creates **default constructor**(with no arguments) for the class.
- And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.
- Default constructor provides the **default values** to the object like **0**, **null**, etc. depending on the type.

2. Parameterized Constructor: A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with your own values, then use a parameterized constructor.

3. Copy Constructor in Java: Like C++, Java also supports copy constructor. But, unlike C++, Java doesn't create a default copy constructor if you don't write your own.

```
class Complex {
   private double re, im;
   Complex(Complex c) {  // copy constructor (Has to be self declared UNLIKE C++)
      System.out.println("Copy constructor called");
      re = c.re;
      im = c.im;
   }
}
```

Q. Does constructor return any value?

A. There are no **return** statements in constructor, but constructor **returns current class instance**. We **can** write **return** inside a constructor.

- Constructor Overoading: Like methods, we can **overload constructors** for creating objects in different ways. Compiler differentiates constructors on the basis of numbers of parameters, types of the parameters and order of the parameters.

Q. How constructors are different from methods in Java?

A. 1. Constructor(s) must have the same name as the class within which it defined while it is not necessary for the method in java.
2. Constructor(s) do not return any type while method(s) have the return type or void if does not return any value.
3. Constructor is called only once at the time of Object creation while method(s) can be called any numbers of time.

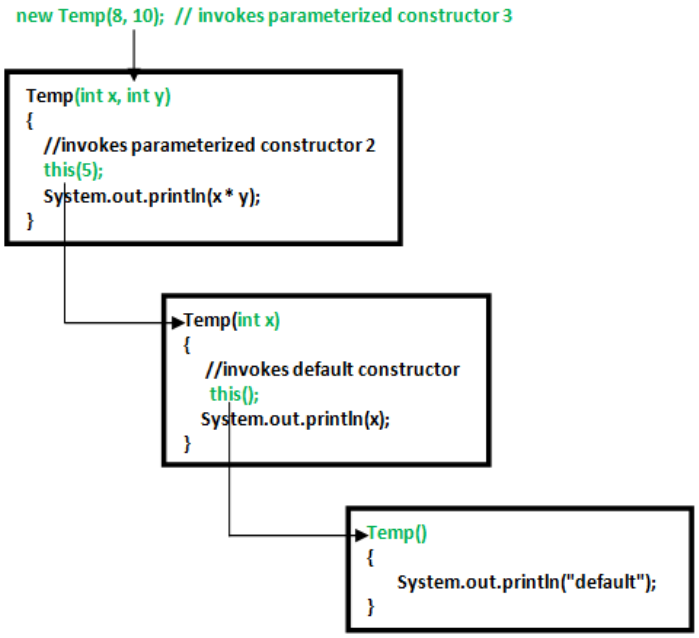
Constructor Chaining - Constructor chaining is the process of calling one constructor from another constructor with respect to current object.

Constructor chaining can be done in **two** ways:

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

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

Constructor Chaining within same class using this() keyword :



Rules of constructor chaining :

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

Constructor Chaining to other class using super() keyword :

- super() must be first line of constructor.
- In case of default / no argument constructors, JVM adds super() before the first line you wrote inside constructor.

Alternative method (using Init block): Initializer block contains the code that is **always executed whenever an instance** is created. It is used to declare/initialize the **common part** of **various constructors** of a class.

- We can note that the contents of initializer block are executed whenever any constructor is invoked (before the constructor's contents).
- The order of initialization constructors and initializer block doesn't matter, initializer block is always executed before constructor.
- Example:

```
class Temp  {
   // block to be excuted first
   {
      System.out.println("init");
   }
   Temp()  {
      System.out.println("default");
   }
   Temp(int x)  {
      System.out.println(x);
   }
   // block to be executed after the first block which has been defined above.
   {
      System.out.println("second");
   }
   public static void main(String args[])  {
      new Temp();
      new Temp(10);
   }
}
```

Output :

```
init
second
default
init
second
10
```

➢ For more information/examples refer to: https://www.geeksforgeeks.org/instance-initialization-block-iib-java/

Static Blocks :

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

➢ This code inside static block is **executed only once**: the first time the **class is loaded** into memory.

➢ Example:

```
// filename: Main.java
class Test {
    static int i;
    int j;

    // start of static block
    static {
        i = 10;
        System.out.println("static block called ");
    }
    // end of static block
}

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);
    }
}
```

Output:

static block called
10