1. What is a Constructor?
Ans:
Constructor is a special method used to initialize objects of a class. Object creation is not enough, it is necessary to initliaze the object after which object is able to respond properly, this task is done by the Constructor method. Whenever we are creating an object some piece of the code will be executed automatically to perform initialization of an object. This piece of code is nothing but a constructor. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes.

Rules for writing a Constructor method:

- Name of the constructor and name of the class must be the same.
- Constructor should have no return type to it as it is not applicable to a Constructor.
- The modifiers applicable for constructors are private,public,protected,default

Example:

```
class Student{
    public Student(){
        System.out.println("Constructor called!");
    }
}
```

2. What is Constructor Chaining?
Ans:
The general rule is that a class can contain many constructors in it. Since all these constructors have the same name(same as the class) and they differ only by the type/number of arguments, this concept is known as Constructor Chaining and the class is hence called “Chained Class”.
Example:

```
class Test {
    Test(double d) {
        System.out.println("double argument constructor");
    }
    Test(int i) {
         this(10.5);
         System.out.println("int argument constructor");
    }
    Test() {
        this(10);
        System.out.println("no argument constructor");
    }
}
```

```
public class MainApp {
    public static void main(String[] args) {
        Test t1= new Test();//double int no argument constructor
        Test t2= new Test(10);// double int argument constructor
        Test t3= new Test(10.5);//double argument constructor
    }
}
```

Note that all constructors have same name but differ in their type and number of arguments.

3. Can we call a subclass constructor from a superclass constructor?

Ans:

First, let me provide a background for understanding **subclass** and **superclass**.
All classes in Java have a superclass from which they are inherited. The **superclass** for all classes is called Object and the class that we created or write is called the **subclass** of the **superclass** Object.

Example:

```
class Test extends Object{
}
```

Here, Test is the class created and is inherited from the Object class. So, Object class is the **superclass** and Test class is the **subclass**.

```
class Test extends Object{
    public Test(){
        super(); // This super method calls Constructor of the Super class,
i.e, Object here
        System.out.println("No-argument Constructor called!");
    }
}
```

In the above code, we are calling the **superclass** Constructor from the **subclass** Test. This is allowed in Java.
But the opposite of it is not possible.
So the answer to the question is:
Meaning that, we cannot call a **subclass** Constructor from the **superclass** Constructor. The reason being, that an instance of a subclass **IS A** instance of the superclass while the vice-a-versa is not true. Here, **IS A** is a concept where 'Foo **IS A** Bar' tells us that an instance of Foo can be held in a reference of Bar. In other words, Bar is an instance of Foo and the vice-versa is not true.
subclass is a class inherited from the superclass and not the other way around.
For example the class Mango is a subclass of the class Fruit. Here we can say that Mango **IS A** Fruit but we cannot say that Fruit **IS A** Mango. So we can call the constructor of Fruit from the constructor of Mango but not the other way round.

If we do so, then the compiler will generate an error:

```
class SuperClass{
     // This is the superclass Constructor
     SuperClass(){
        System.out.println("SuperClass constructor");
        SubClass(); // Here, we are calling the subclass Constructor
    }
}
public class SubClass extends SuperClass
{
    // This is the subclass constructor
    SubClass (){
        System.out.println("Subclass constructor");
    }
      public static void main(String[] args) {
            System.out.println("Constructor test");
      }
}
```

The output we'll get will be:

```
SubClass.java:11: error: cannot find symbol
        SubClass();
        ^
  symbol:   method Main()
  location: class SuperClass
1 error
```

This indicated that we cannot call a subclass constructor from the superclass.

4. What happens if you keep a return type for a constructor?

Ans:

Ideally, no return type is allowed for a constructor. By definition, if a method has a return type, it's not a constructor and it won't be result in a compiler error, instead it will be treated as a normal method by the Java language. But compiler gives a warning saying that method has a constructor name.

Example:

```
class A{
    public int A(){
        return 0;    // Warning generated for the return type here
    }
}
```

Warning Generated: This method has a Constructor name.

5. What is No-arg constructor?
Ans:
A constructor that has no parameter is known as the No-argument or No-arg or Zero argument constructor. If we don't define a constructor in a class, then the compiler creates a 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.
Example:

```
class Test {
    public Test() {
        System.out.println("No-arg Constructor called");
    }
}

public class Launch {
    public static void main(String[] args) {
        Test t = new Test(); // On executing this line output will be:
No-arg Constructor called
    }
}
```

6. How is a No-argument constructor different from the default Constructor?
Ans:
Default Constructor is like a No-argument constructor but it is created by the Java Compiler automatically at the time of object creation if no constructor is specified/written in the class. It will be invisible. The function of Default constructor is to initialize default values to the object variables.
The special feature of Default Constructor is such that: if a no-argument or parameterized constructor is written in the class, then the compiler does not create a default constructor and the default constructor is taken out and is invalid. If we try to call the default constructor in such a case then the compiler will throw an error. That default constructor is being overloaded and now acts like a parameterized constructor.
The default constructor is a no-args constructor that the Java compiler inserts on your behalf; it contains a default call to super(); which is the default behavior. Also, the body of a Default Constructor will always remain empty. If we implement any constructor then we no longer receive a default constructor.
No-argument Constructor is created by the programmer and it is written explicitly inside the class. In the No-argument Constructor, which takes no arguments, we can have any kind of functionality or code inserted in it since it is written by the programmer but the same is not with the case of Default constructor. Also, once a No-argument constructor is declared/written in the class, the compiler automatically disallows its Default Constructor to be called and it will allow

the use of only this No-argument Constructor declared. So, the conclusion is that every Java class will have a compiler generated Default Constructor **or**(a big or) a programmer written Constructor but not both simultaneously. In other words, a Default Constructor is a particular kind of No-argument Constructor.
Example:
These two classes are identical in terms of behavior (both have a no-argument constructor that does nothing), but one has a default constructor and the other does not.

```
class NoDefaultConstructor {
    NoDefaultConstructor() {
        // This is a no-argument constructor with an empty body, but is not a default constructor.
}
```

```
class HasDefaultConstructor {
    // This class has no declared constructors, so the compiler inserts a default constructor.
}
```

In each case, we can create an instance with `new NoDefaultConstructor()` or `new HasDefaultConstructor()`, because both classes have no-argument constructors, but only `HasDefaultConstructor`'s no-argument constructor is a default constructor.

7. When do we need Constructor Overloading?
Ans:
Constructor Overloading enables a Java programmer to create many constructors with same name but different arguments/types for creating an object of the class in several ways. Based on the requirement, we can create as many constructors as we want which enables the creation of an object in different ways.
The key advantages of making use of constructor overloading while writing Java programs are:
i. The class instances can be initialized in several ways with the use of constructor overloading.
ii. It facilitates the process of defining multiple constructors in a class with unique signatures.
iii. Each overloaded constructor performs various tasks for specified purposes.

Example:
We are working on a Box() class to calculate the volume of a box depending on the specifications given to it. If 3 parameters are passed to create an object of the Box class then we can calculate the volume as length*breadth*height as it is a cuboid now. If only one parameter is passed to create the Box class then we can calculate the volume as length*length*length as it becomes a cube now. Suppose we want to create a Box object with no initial dimensions, then we can create a Constructor method which takes no arguments and returns the volume as 0. All the above 3 methods can be performed buy using the Concept of

Constructor overloading.

```
// Java program to illustrate
// Constructor Overloading
class Box {
      double width, height, depth;
      // constructor used when all dimensions
      // specified, i.e, 3 parameters are passed
      Box(double w, double h, double d) {
            width = w;
            height = h;
            depth = d;
      }
      // constructor used when no dimensions
      // specified, i.e, no parameters are passed
      Box(){
            width = height = depth = 0;
      }

      // constructor used when cube is created, i,e only
            // only one parameters is passed
      Box(double len) {
            width = height = depth = len;
      }
      // compute and return volume
      double volume() {
            return width * height * depth;
      }
}

public class Test {
      public static void main(String args[]) {
            // create boxes using the various
            // constructors
            Box mybox1 = new Box(10, 20, 15);
            Box mybox2 = new Box();
            Box mycube = new Box(7);

            double vol;

            // get volume of first box
            vol = mybox1.volume();
            System.out.println(" Volume of mybox1 is " + vol);
            // get volume of second box
            vol = mybox2.volume();
            System.out.println(" Volume of mybox2 is " + vol);
            // get volume of cube
            vol = mycube.volume();
            System.out.println(" Volume of mycube is " + vol);
      }
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is 0.0
Volume of mycube is 343.0
```

8. What is Default constructor? Explain with an Example.
Ans:
For every java class constructor concept is applicable.
If we don't write any constructor, then the compiler will generate a default constructor.
If we write at least one constructor then the compiler won't generate any default constructor, so we say every java class will have a compiler generated default constructor or programmer written constructor but not both simultaneously.
Default Constructor is like a No-argument constructor but it is created by the Java Compiler automatically at the time of object creation if no constructor is specified/written in the class. It will be invisible. The function of Default constructor is to initialize default values to the object variables.
The special feature of Default Constructor is such that: if a no-argument or parameterized constructor is written in the class, then the compiler does not create a default constructor and the default constructor is taken out and is invalid. If we try to call the default constructor in such a case then the compiler will throw an error. That default constructor is being overloaded and now acts like a parameterized constructor.
The default constructor is a no-args constructor that the Java compiler inserts on your behalf; it contains a default call to super(); which is the default behavior. Also, the body of a Default Constructor will always remain empty. If we implement any constructor then we no longer receive a default constructor.

Example:
In the below code constructor is not written for the class Test

```
class Test{
    private int age;
    private String name;
```

```java
    public void display(){
        System.out.println(age + " " + name);
    }
}

public class Launch{
    public static void main(String[] args){
        Test t = new Test();
    }
}
```

But Java language requires a Constructor to initialize the object variables. So, behind the scenes, the Compiler will add the Default Constructor. Hence, the code will actually look like the following during compilation:

```java
class Test{
    private int age;
    private String name;

    // This code is used behind the scenes by the Java compiler
    public Test(){
        // Since it is a default constructor it will not have any body in
it
    }
    public void display(){
        System.out.println(age + " " + name);
    }
}

public class Launch{
    public static void main(String[] args){
        Test t = new Test();
    }
}
```