Constructors in Java

Constructor is a block of code that initializes the newly created object. A constructor resembles an instance method in java but it is not a method as it does not have a return type.

Constructor has the same name as the class as shown below:

```
public class SimpleClass
{
        /* Note: The name of the constructor
          * matches the name of the class
          */
        SimpleClass();
}
```

To understand the working of a constructor, let's look at an example below:

SimpleClass **obj** = **new** SimpleClass();

The new keyword creates the object of class SimpleClass. It also invokes the constructor to initialize this newly created object. Look at the code below to get more understanding.

```
public class Welcome
{
        String message;          //instance variable

         Welcome()               //Constructor
        {
                this.message = "Welcome to the QOL Java Programming Class";
        }

         public static void main(String[] args)
        {
                Welcome obj = new Welcome();
                System.out.println(obj.message);
         }
}
```

Types of Constructors

There are 3 types of constructors, namely: **Default**, **No-arg**, **Parameterized**
If you do not implement any constructor in your java class, Java provides a default constructor for you.
If you provide a constructor then you do not get a default constructor from java.

no-arg constructor:

Constructors with no arguments is known as **no-arg constructor**. It is the same as **default constructor**, however body of the no-arg constructor **can have any code**. The body of the default constructor is **empty**.

Example: no-arg constructor

```
class Testing
{
    public Testing()
    {
       System.out.println("This is a no argument constructor");
    }
    public static void main(String args[])
   {
           Testing obj = new Testing();
    }
}
```

Parameterized constructor

Constructors with arguments (or parameters) are known as Parameterized constructors. Please look at the example below:

```
public class Employee
{
      int empId;                          // instance variable
      String empName;                     // instance variable

      //parameterized constructor with two parameters
      Employee(int id, String name)
      {
            this.empId = id;
            this.empName = name;
      }
      void info()
      {
            System.out.println("Id: "+empId+" Name: "+empName);
      }

      public static void main(String args[])
      {
            Employee obj1 = new Employee(12345,"Johnson");
            Employee obj2 = new Employee(67890,"Margaret");
            obj1.info();
            obj2.info();
      }
}
```

What is the problem in the code below:

```
class Problem
{
    private int x;                  //instance variable

    public Problem (int num)
    {
        x = num;
    }
    public int getValue()
    {
       return x;
    }

    public static void main(String args[])
    {
        Problem myobj = new Problem();
        System.out.println("Value of x → "+myobj.getValue());
    }
}
```

Method Overloading

Method Overloading allows a class to have **more than one method with the same name** (if the argument list is different).

Argument list refers to the **parameters** that a method has. For example, consider the two statements below:

Add(int a, int b)
Add(int a, int b, int c)

Looking at the two methods above, we can make a statement that the argument list of Add(int a, int b) is **different** than the argument list of Add(int a, int b, int c).

General Rules for overloading a method

In order to overload a method, the argument list of the methods must differ in either of these:

1. Number of parameters
2. Data type of parameters
3. Sequence of Data type parameters

For option 1:
add(int, int)
add(int, int, int)

For option 2:
add(int, int)
add(int, float)

For option 3:
add(int, float)
add(float, int)

Points to Note:
Static Polymorphism is also known as compile time binding or early binding
Static binding happens at compile time. Method overloading is an example of static binding because binding of method call to its definition happens at compile time.
Also Note, that Method overloading is an example of Static Polymorphism. We will learn about polymorphism in another lecture.

Inheritance

The process by which one class acquires the properties (data memebers) and functionalites (methods) of another class is called inheritance. The aim of inheritance is to provide reusability of code so that a class has to write only the unique features and rest of the common properties and functionalities can be executed from another class.

Terminology:

Child class: The class that extends the features of another class. It is also known as derived class or subclass.
Parent class: The class whose properties and functionalities are used (inherited) by another class. It is also known as parent class, super class or Base class.

To inherit a class we use the keyword **extends**.
Example:

```
class Teacher
{
    String designation = "Teacher";
    String collegeName = "Emory";

    void does()
    {
        System.out.println("Teaching");
    }
}

public class PhysicsTeacher extends Teacher
{
    String mainSubject = "Physics";

    public static void main(String args[])
    {
        PhysicsTeacher obj = new PhysicsTeacher();
        System.out.println(obj.collegeName);
        System.out.println(obj.designation);
        System.out.println(obj.mainSubject);
        obj.does();
    }
}
```

Please Note: The derived class inherits all the members and methods that are declared as public or protected. If the members or methods of super class are declared as **private** then the derived class cannot use them directly. The private members can obly be accessed using public or **protected getter and setter methods of the super class**.

```
class Teacher
{
    private String designation = "Teacher";
    private String collegeName = "Emory";

    public String getDesignation()
    {
        return designation;
    }

    protected void setDesignation(String designation)
    {
        this.designation = designation;
    }

    protected String getCollegeName()
    {
        return collegeName;
    }

    protected void setCollegeName(String collegeName)
    {
        this.collegeName = collegeName;
    }

    void does()
    {
        System.out.println("Teaching");
    }
}

public class JavaExample extends Teacher
{
    String mainSubject = "Physics";

    public static void main(String args[])
    {
        JavaExample obj = new JavaExample();

        /* Note: we are not accessing the data members
         * directly. we are using public getter method
         * to access the private members of parent class
         */

        System.out.println(obj.getCollegeName());
        System.out.println(obj.getDesignation());
        System.out.println(obj.mainSubject);
        obj.does();
      }
}
```

Inheritance and Method Overriding

When we declare the same method in a child class which is already present in the parent class, it is called **method overriding**. If we call that method, it will come from the child class. However, we can call the parent class method by using the **super** keyword.

Example:

```
class ParentClass
{
    //Parent class constructor
    ParentClass()
    {
        System.out.println("Constructor of Parent");
    }

    void disp()
    {
        System.out.println("Parent Method");
    }
}

class JavaExample extends ParentClass
{
    JavaExample()
    {
        System.out.println("Constructor of Child");
    }

    void disp()
    {
        System.out.println("Child Method");
        //Calling the disp() method of parent class
        super.disp();
    }

    public static void main(String args[])
    {
        //Creating the object of child class
        JavaExample obj = new JavaExample();
        obj.disp();
    }
}
```

Arrays

An array is used to store a collection of data. Actually, think of an array as a collection of variables of the same type. Instead of declaring several variable to represent different values of the same data type, one can use an array.

Syntax:
datatype[] arrayRefVar; // preferred way
datatype arrayRefVar[]; // right bot not preferred.

Example:
double[] myarray = new double[10];   // preferred way
//double myarray[] = new double[10];   ← It works but not preferred

String citynames[] = {“Chicago”,”Detroit”,”Houston,”Dallas”,”Atlanta”};

You can create arrays of int, short, double, String, Char, etc.

ArrayList

It iused because of the functionality and flexibility it offers. ArrayList can dynamically grow or shrink but Arrays cannot. There are many ways to initialize the arraylist.

1. Initialization using Array.asList
2. Initialization using Anonymous inner class (this will not be covered)
3. Normal way of Arraylist initialization 🡸 Mostly used
4. Using Collection.ncopies (this will not be covered)

Arraylist Example: Method 1

Syntax: ArrayList<Type> obj = new ArrayList<Type>(Arrays.asList(Object o1, Object o2, Object o3, ...so on));

```
import java.util.*;
public class InitializationExample1
{
            public static void main(String args[])
            {
               ArrayList<String> ArrayList1j = new ArrayList<String>(Arrays.asList("Paul", "Peter", "Somebody"));
              System.out.println("Elements are: " +ArrayList1);
            }
}
```

Arraylist Example: Method 3

```
Syntax:  ArrayList<T> obj = new ArrayList<T>();
         obj.add(Object 1);
         obj.add(Object 2);
         obj.add(Object 3);
         …
         …
```

```
Import java.util.*;
Public class initializationExample3
{
            Public static void main(String args[])
            {
                        ArrayList<String> books = new ArrayList<String>();
                        books.add(“Java Book”);
                        books.add(“C# Book”);
                        books.add(“SomeOther Book”);
                        System.out.println(“Following books are stored in this array list:\n” +books);
            }
}
```

Detailed Example of ArrayList Method 3:

```
public class ArrayListExample
{
        public static void main(String args[])
        {
            /*Create an ArrayList: of type Sting */
             ArrayList<String> myList = new ArrayList<String>();

             /*This is how elements should be added to the array list*/
             myList.add("Steve");
             myList.add("James");
             myList.add("Richard");

             /* Displaying array list elements */
             System.out.println("Members in my List: "+myList);

             /*Add element at the given index*/
             myList.add(0, "Lincoln");
             myList.add(1, "Patrick");

             /* Displaying the changed array list elements */
             System.out.println("Members in my List: "+myList);

             /*Remove elements from array list */
             myList.remove("Steve");
             myList.remove("Richard");

             System.out.println("Current array list is: "+myList);

             /*Remove element from the given index*/
             myList.remove(1);

             System.out.println("Current array list is: "+myList);
        }
}
```

Example using Integer

```
import java.util.ArrayList;
public class Details
{
        public static void main(String [] args)
        {
                ArrayList<Integer> intList = new ArrayList<Integer>();
                System.out.println("Initial size of the List : "+intList.size());

                /* Add elements to the list */
                intList.add(1);
                intList.add(13);
                intList.add(45);
                intList.add(44);
                intList.add(99);

                System.out.println("Size after few additions: "+ intList.size());

                /* Remove elements from the list */
                intList.remove(1);
                intList.remove(2);

                System.out.println("Size after remove operations: "+ intList.size());

                System.out.println("Final ArrayList: \n\n");

                /* Notice the Enhanced loop */
                for(int num: intList)
                {
                        System.out.println(num);
                }
        }
}
```