Object Oriented Programming Concepts

Unit- II

Topics to be Covered...

- POP Vs. OOP
- Basics of OOP
- Class, Field, Methods, Object
- Constructors
- Method Overloading
- Constructor Overloading
- Accessing Rules
- This Keyword, Final Keyword and Static Keyword
- String Class
- User Input
- Wrapper Class

POP v/s OOP

CRITERIA	POP	OOP
Divided Into	In POP, program is divided into small parts called functions.	In OOP, program is divided into parts called objects.
Importance	In POP, Importance is not given to data but to functions as well as sequence of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it is more realistic.
Approach	POP follows Top-Downapproach.	OOP follows Bottom-Up approach.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data can not move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is less secure.	OOP provides Data Hiding so provides more security.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are : C, VB, FORTRAN, Pascal.	Example of OOP are : C++, JAVA, VB.NET, C#.NET.

Pillars of OOP

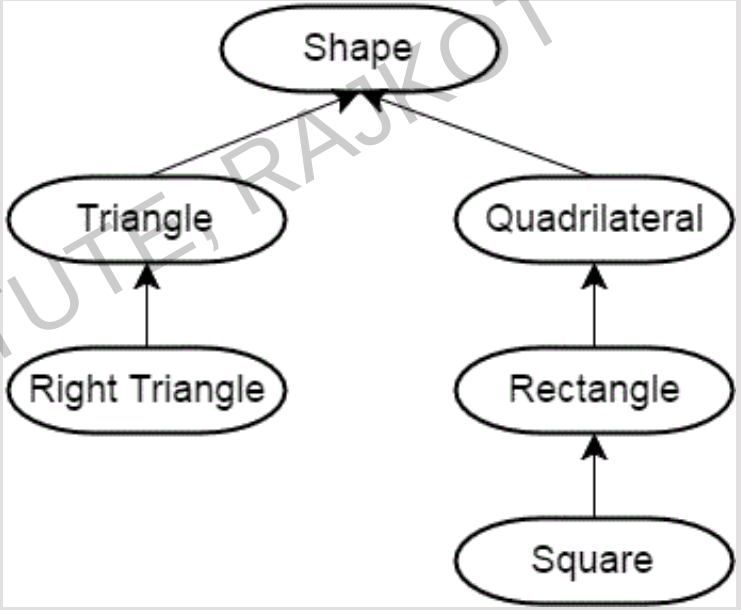
- Class & Object:
  - Class:
    - User defined Data type
    - Blueprint, a Passive Entity
    - Example: Blueprint & plan of the building
  - Object:
    - Instance of a class.
    - Runtime Entity, occupies space in memory.
    - It has data and set of functions that it can execute (behaviour).
    - Example: Actual building built using the blueprint and material (Data).

Pillars of OOP

- Abstraction:
  - Hiding differences and finding similarities between objects to gather them under the same roof (class) and reduce complexity.
  - Example: Car is considered as a single object with its behaviour and not like a group of small parts.
- Encapsulation:
  - Wrapper to bind data and code inside it preventing them to be accessed arbitrarily by other code outside that wrapper.
  - It hides the implementation complexity from outside code just giving them what they need.
  - Example: Capsule of medicine.

Pillars of OOP

- Inheritance:
  - Hierarchical classification of classes according to need.
  - Code reusability.
  - Example: Shape hierarchy
- Polymorphism:
  - Greek words Poly meaning “Many” & Morphs meaning “Masks” (Behaviour).
  - One interface, Multiple methods.
  - Example: Dog smelling food, Dog smelling a Cat



Pillars of OOP

- Dynamic Binding:
  - Connecting the method with its body is known as binding.
  - Static binding is done at compile time.
  - But dynamic binding is done at run time which provides polymorphism.
- Message Passing:
  - Objects can communicate.
  - Not freely but using set of well defined methods to send and receive information to-from each other.
  - This prevents free movement of data outside the object.

Class

- Class is a blueprint or a template for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.
- So, a class is a template for objects, and an object is an instance of a class.
- When the individual objects are created, they inherit all the variables and methods from the class.
- Everything in Java is associated with classes and objects, along with its attributes and methods.
- Example: Your Car is an object of a GenericCar class. The car has attributes, such as weight and color, and methods, such as drive and brake.

Class

- Java classes can be defined using following syntax:

```
<access-specifier> class <ClassName> {
        <access-specifier> <other-modifiers> <data-type> varName1;
        <access-specifier> <other-modifiers> <data-type> varName2;
        ...
        <access-specifier> <other-modifiers> <return-type> methodName1 (<arguments>) {
                //Method Statements.
        }
        <access-specifier> <other-modifiers> <return-type> methodName2 (<arguments>) {
                //Method Statements.
        }
        ...

}
```


Objects

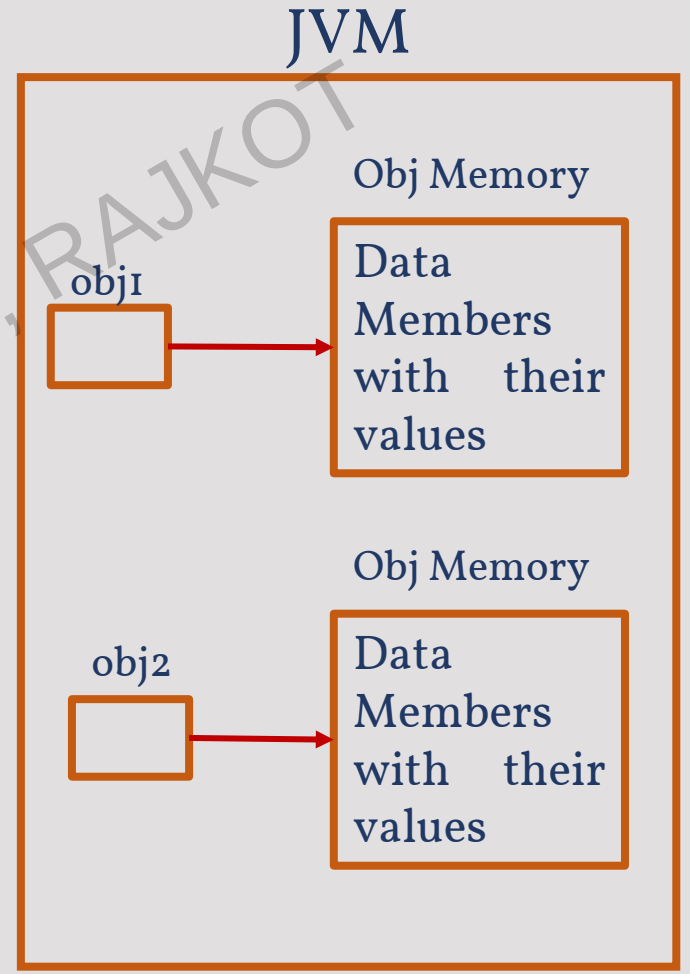
- Java objects can be defined using following syntax:

```
<ClassName> <objectName> = new <ConstructorName>(<argumetns>);
```

1. Object Declaration:
Here, compiler is just informed that an object of the class is declared.
No memory allocation is done.

2. Object Instantiation:
Keyword used to allocate new memory inside JVM to the object declared.

3. Object Initialization:
One of the constructor used to initialize all the members of an object that is declared here.



Example

```
class York {
	int x, y;
	public void setXY(int a, int b) {
		x = a;
		y = b;
	}
	public void printXY() {
		System.out.println("X: " + x + "Y: " + y);
	}
}
```


Example

```
public class MainClass {
	public static void main(String[] args) {
		York ny = new York();
		ny.x = 10; // Initialize x
		ny.y = 20; // Initialize y
		ny.setXY(10, 20); // Initialize x, y through setter method
		ny.printXY();
	}
}
```


Methods

- Methods are blocks of code that perform specific tasks and are associated with classes and objects.
- Methods are also known as functions in other programming languages.
- A method is a block of code which only runs when it is called, you can also pass data while calling a method that is known as parameters.
- Why use methods?
  - Code Organization
  - Code Reusability
  - Modularity
  - Parameter Passing
  - Etc.

Methods

- Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions.
- It is defined with the name of the method, followed by parentheses ().
- A method must be declared within some class.
- Syntax:

```
<access-modifier> <other-modifiers> <return-type> <methodName> (<type1> <param1>, …) {
        //Method Definition (Body of the method)
        //Contains code to be executed when the method is called.
}
```


Methods

- Example:

```
public class Foo {
        public static int myMethod(int a, String b) {
                System.out.println(b + " : " + a); //Method body
                return 0;
        }
}
```

- Here, myMethod() is the name of the method, static means that the method belongs to the class and not an object of the class and int means that this method returns an int value hence the return statement must be the part of method body.

Methods

- To call a method in Java, write the method's name followed by parentheses( ) and a semicolon, you can call multiple times also as per requirements.
- Syntax:

```
public class MethodDemo1 {
        void myMethod() {
                System.out.println("I just got Executed!");
        }
        public static void main(String[ ] args) {
                MethodDemo1 obj = new MethodDemo1();
                obj.myMethod();
        }
}
```

```
Output
I just got Executed!
```


Methods

- Information can be passed to methods as parameter and it act as variables inside the method.
- Parameters are specified after the method name, inside the parentheses, you can add as many parameters as you want, just separate them with a comma.
- Example:

```
public class MethodDemo2 {
        void greet(String n) {
                System.out.println("Hello, " + n);
        }
        public static void main(String[ ] args) {
                MethodDemo2 obj = new Methoddemo2();
                obj.greet("4th C");
        }
}
```

```
Output
Hello, 4th C
```


Methods

- Multiple Parameters can also be passed

```
public class MethodDemo3 {
	void greet(String n, int age) {
		System.out.println("Name: " + name + "\nAge: " + age);
	}
	public static void main(String[ ] args) {
		MethodDemo3 obj = new MethodDemo3();
		obj.greet("John", 20);
	}
}
```

```
Output
Name: John
Age: 20
```


Methods

- Methods can also have return values that can be stored after the execution of the method into a variable by caller of the method.

```
public class MethodDemo4 {
	int sum(int a, int b) {
		return (a + b);
	}
	public static void main(String[ ] args) {
		MethodDemo4 obj = new MethodDemo4();
		int s = obj.sum(10, 20)
		System.out.println("Sum is: " + s);
	}
}
```

```
Output
Sum: 30
```


Method Overloading

- Method overloading can be done when the methods are made with same name but it has different signatures.
- Signature consists of number & type of the arguments.

```
class Addition {
      public int add(int a, int b) {
            return (a + b);
      }
      public int add(int a, int b, int c) {
            return (a + b + c);
      }
      public String add(String a, String b) {
            return (a + b);
      }
}
```

Ovreloaded Method add()

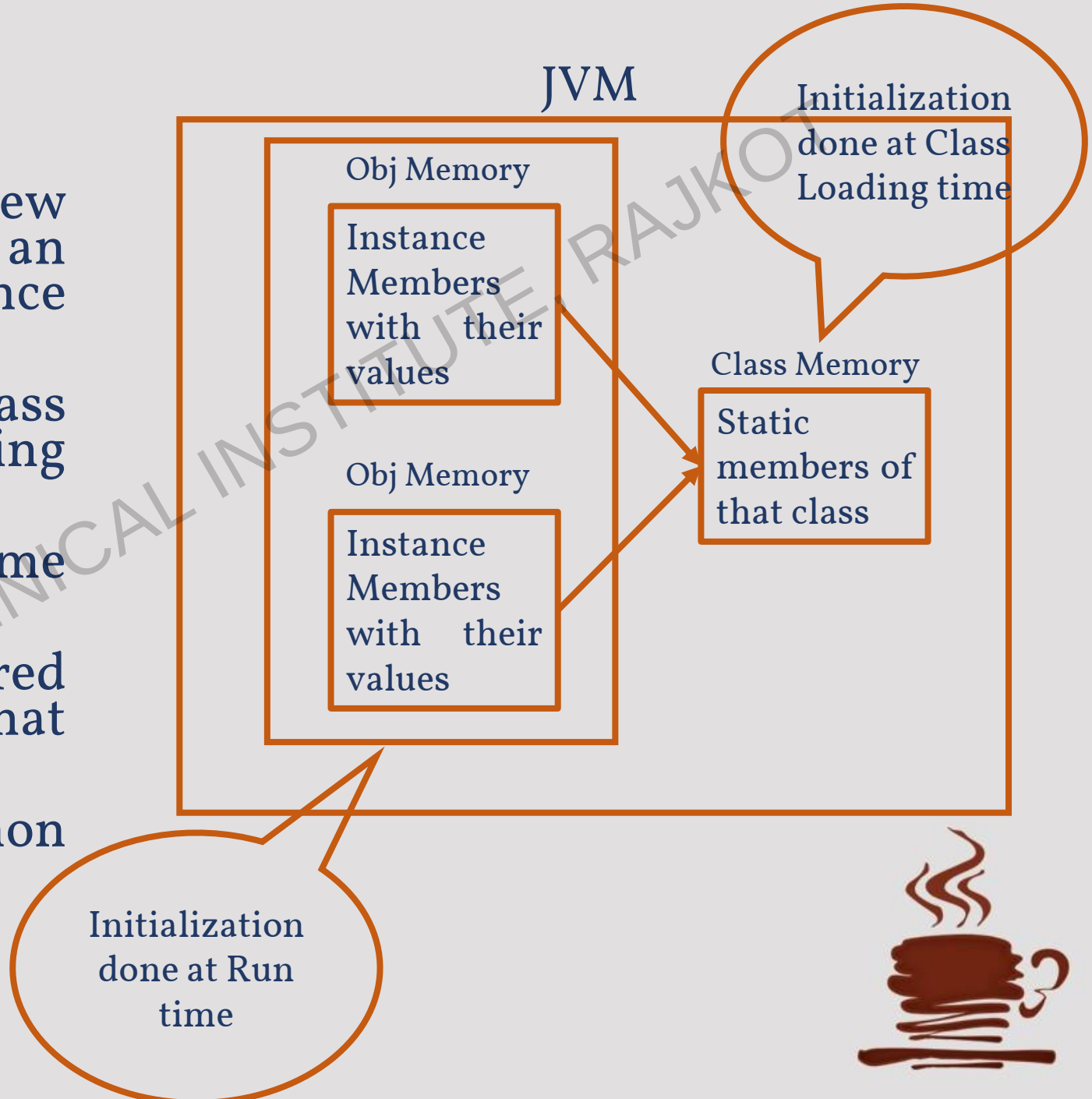
Accessing Rules

- Access control is the process of controlling visibility of a variable or method.
- There are four levels of visibility :
  1. **public:** member is accessible from any other class.
  2. **private:** member is accessible only within the same class
  3. **protected:** member is accessible within the same package and by subclasses
  4. **default (no modifier):** member is accessible only within the same package

Access Modifier	Same Class	Same Package	Other Subclass	Other Package
private	Y	N	N	N
default (Friendly)	Y	Y	Same Package = Y Other Package = N	N
protected	Y	Subclass = Y Other Class = N	Y	N
public	Y	Y	Y	Y

static keyword

- Members that are allocated new space with a new instantiation of an object are known as Instance Variables.
- static keyword is used to make class members. So they are accessed using class name.
- Initialization is done at compile time only.
- One copy of member is shared between all the instances of that class.
- Static members cannot access non static members of the class.



Example

- Example of counting number of objects.

Static Blocks

- These are used to initialize static fields or to perform any static initialization required by the class before any kind of class instantiation.
- They are executed only once when the class is loaded into memory.

```
public class StaticDemo {
	static {
		//This static block will be executed when the class is loaded
		System.out.println("Inside static block");
	}
	public static void main(String[] args) {
		System.out.println("Inside main method");
	}
}
```

```
Output
Inside Static Block
Inside Main Method
```


final keyword

- Can be used with variable/objects, methods and classes.
- If used with variable it makes the variable constant.
- If used with method then it does not allow that method to be overridden.
- If used with class then it would not allow that class to be inherited further.

```
class Foo {
      final int x = 10; //Initialization done and it is only allowed
once, either here or in the constructor.
      public void increment() {
            x ++; //Generates an error.
      }
}
```


this keyword

- this refers to the current object.
- When we are required to refer to the object which is currently getting executed, this can be used.

```
public class ThisDemo {
	public void show() {
		System.out.println(this);
	}
	public static void main(String[] args) {
		ThisDemo obj = new ThisDemo();
		obj.show()
		System.out.println(obj);
	}
}
```

```
Output
ThisDemo@3b846d56
ThisDemo@3b846d56
```


this keyword

- It can also be used to remove the confusion in compiler between variables inside an instance method, if the local variable name in a method is same as one of the instance member of the class then this can be used to refer to the instance member.

```
public class ThisDemo2 {
        int a;
        public void setA(int a) { this.a = a; }
        // Here this.a -> Instance Member & a -> Local Variable
        public static void main(String[] args) {
                ThisDemo2 obj = new ThisDemo2();
                obj.setA(20);
                System.out.println(obj.a);
        }
}
```

```
Output
20
```


Recalling this Example

```
class York {
	int x, y;
	public void setXY(int a, int b) {
		x = a;
		y = b;
	}
	public void printXY() {
		System.out.println("X: " + x + "Y: " + y);
	}
}
```


Going towards the Concept of Constructor

- In previous example you need to call the method setXY( ) to initialize the members to their values.
- May be someone can forget to do that because compiler does not force you to call that method before using that object “ny” any further.
- Constructor helps you with that.
- Definition: Constructor is a special member function of a class without return type (not even void) with the name same as class name which is called only while object is getting initialized and it is used to perform initialization of all the members of the class.
- If no constructor is written then java provided a default constructor but if we declare any constructor then the default one won’t work.
- Default constructor initializes all the members in java to their default values as per its type.

Default Values Revised

Data Type	Default Value
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
Reference type (Objects)	null

Example

```
class York {
      private int x, y; // Members are always made private. Unless
some special case.
      public York(int a, int b) {
            x = a;
            y = b;
      }
      public void printXY() {
            System.out.println("X: " + x + "Y: " + y);
      }
}
```

```
public void setXY(int a, int b)
{
            x = a;
            y = b;
} // This method is now Optional
```


Example

```
public class MainClass {
	public static void main(String[] args) {
		York ny = new York(10, 20);
		ny.printXY();
	}
}
```


Constructors Continued...

- Types of Constructors:
  - Default Constructor
  - Parameterized Constructor
  - Copy Constructor
  - Constructor Overloading

Constructor Overloading

- Constructor overloading in Java allows a class to have multiple constructors with different parameter lists.
- This enables objects of the class to be initialized in different ways based on the arguments provided.
- Constructor overloading follows the same principles as method overloading, where methods with the same name but different parameters are declared within the same class.

Default & Parameterized Constructors

```
class Rectangle {
      int w, h;
      Rectangle() { w = 1; h = 1; } // Default Constructor
      Rectangle(int w) { this.w = w; h = w; } // Parameterized Constructors
      Rectangle(int w, int h) {
            this.w = w;
            this.h = h;
      }
}
```

Overloaded Constructors

Revisiting this Keyword

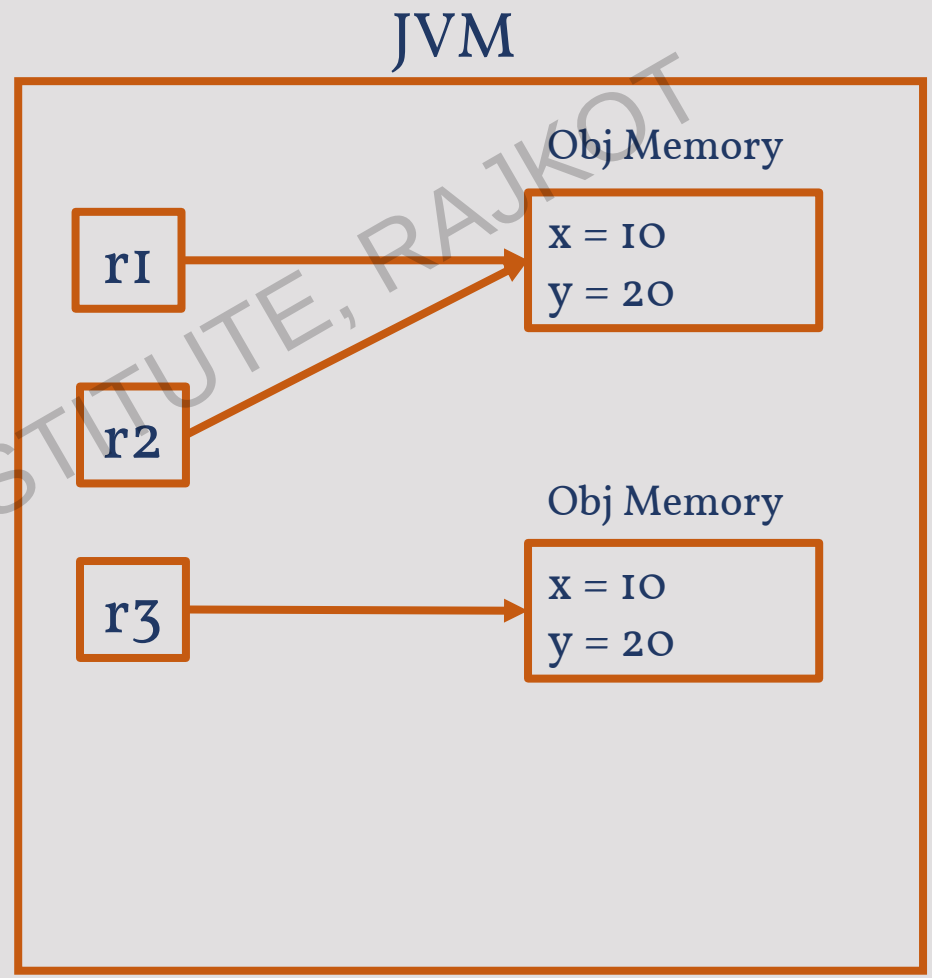
- this can be used in constructor to call some other constructor of that class.

```
public class ThisDemo3 {
        int a, b;
        ThisDemo3(int a) { this.a = a; b = 0;}
        ThisDemo3(int a, int b) {
                this(a);
                // Calling another constructor of this class from this Constructor
                // Next lines of code will be executed as it is.
                this.b = b;
        }
        public static void main(String[] args) {
                ThisDemo3 obj = new ThisDemo3(10);
                ThisDemo3 obj = new ThisDemo3(20, 30);
                System.out.println("obj1.a: " + obj1.a + "\tobj1.b: " + obj1.b);
        System.out.println("obj2.a: " + obj2.a + "\tobj2.b: " + obj2.b);
        }
}
```

```
Output
obj1.a: 10 obj1.b: 0
obj2.a: 20 obj2.b: 30
```


Copy Constructor

```
Rectangle(Rectangle r1) {
      w = r1.w;
      h = r1.h;
}
```



- Shallow Copy v/s Deep Copy
  - `Rectangle r1 = new Rectangle (10, 20);`
  - `Rectangle r2 = r1; //Shallow Copy`
  - `Rectangle r3 = new Rectangle (r1); //Deep Copy`

Passing Object as an Argument

- In Java, you can pass objects as parameters to methods just like you pass primitive data types.
- When you pass an object as a parameter, you are actually passing a reference to the object, not a copy of the object itself. This means that changes made to the object within the method will affect the original object.

Passing Object as an Argument

```
class Rectangle {
      int w, h;
      Rectangle(int w, int h) { this.w = w; this.h = h; }
      public void change(Rectangle r) { r.w = 20; }
      public static void main(String[] args) {
            Rectangle r1 = new Rectangle(2, 3);
            Rectangle r2 = new Rectangle(1, 1);
            System.out.println("Before r1.w: " + r1.w);
            r2.change(r1);
            //Passed by reference, hence r1 gets updated.
            System.out.println("After r1.w: " + r1.w);
}     }
```

```
Output
Before r1.w: 2
After r1.w: 20
```


User Input: Scanner Class

- Various Stream classes are available for reading/writing to the input/output streams.
- Scanner is one of the utility class available in java.util package, capable of reading input stream (i.e. getting data from console at run-time).
- First of all the Scanner class needs to be imported to current environment
- Then a Scanner type object is required.
- Calling various methods available in Scanner using its object will help in reading user inputs.

User Input: Scanner Class

```
import java.util.Scanner; //Importing Scanner Class

public class ScannerDemo {
	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in)
		//Passing an Input Stream.
		System.out.print("Enter username: ");
		String userName = sc.nextLine(); //1 of the Scanner Method
		System.out.println("Username is: " + userName);
	}
}
```

```
Output
Enter Username: admin
Username is: admin
```


User Input: Scanner Class

Method	Description
`public boolean nextBoolean()`	Reads a boolean value from the user
`public byte nextByte()`	Reads a byte value from the user
`public short nextShort()`	Reads a short value from the user
`public int nextInt()`	Reads a int value from the user
`public long nextLong()`	Reads a long value from the user
`public float nextFloat()`	Reads a float value from the user
`public double nextDouble()`	Reads a double value from the user
`public String nextLine()`	Reads a String value from the user

User Input: Scanner Class

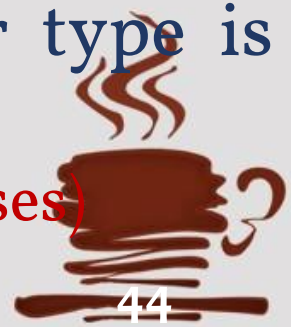
```
import java.util.Scanner;

public class ScannerDemo2 {
	public static void main(String[] args) {
		Scanner myObj = new Scanner(System.in);
		System.out.println("Enter name, age and salary:");
		String name = myObj.nextLine();  // String input
		int age = myObj.nextInt();       // Numerical input
		double salary = myObj.nextDouble();
		System.out.println("Name: " + name);
		System.out.println("Age: " + age);
		System.out.println("Salary: " + salary);
	}
}
```

```
Output
Enter name, age and salary:
Roy
22
56000
Name: Roy
Age: 22
Salary: 56000.0
```


User Input: Command Line Arguments

- In Java, you can also accept user input using command-line arguments. i.e. arguments passed to main() method. Command-line arguments are passed as an array of strings String args[ ].
- Any number of command line arguments can be passed to the program and command line arguments are be received by the String array of main method.
- String is used as its type because from all the primitive types supported by Java, String has the largest accepted charset, hence we can pass any data as an argument.
- Further parsing can be done on those arguments if any other type is required.
  - Ex: String -> int, String -> byte, String -> double etc. (Using Wrapper classes)



User Input: Command Line Arguments

```
public class CommandLnExample {
	public static void main(String[] args) {
		System.out.println("Name: " + args[0]);
		System.out.println("Age: " + args[1]);
	}
}
```

Command Line Arguments

```
Output
_> javac CommandLnExample
_> java CommandLnExample Tom 20
Name: Tom
Age: 20
```

String Class

- String is a sequence of characters enclosed within double quotes. (Ex. "Sample String"), but, unlike many other languages that implement strings as character arrays, Java implements strings as objects of String class.
- Immutable: Once a String object has been created, you cannot change the characters that comprise that string, each time you need an altered version of an existing string, a new String object is created that contains the modifications and the original string is left unchanged.
- There are two ways to create String object:
  - From a String Literal
  - Using String Constructor

String Class

- In Java, you can create a string using a string literal, which is a sequence of characters enclosed in double quotes.
- When you create a string using a string literal, Java checks the "string pool" to see if an identical string already exists. If it does, Java returns a reference to the existing string, rather than creating a new object. This is called string interning.
- If the string does not exist in the pool, a new String object is created and placed in the pool.
- Example:

```
String str1 = "Hello";
```


String Class: Empty and Null Strings

- The empty string "" is a string of length `0`. You can test whether a string is empty by calling

```
if (str.length() == 0) or if (str.equals(""))
```

- An empty string is a Java object which holds the string length (namely, 0) and an empty contents. However, a String variable can also hold a special value, called null, that indicates that no object is currently associated with the variable. To test whether a string is null, i.e. its object is not even initialized.

```
if (str == null)
```

- Sometimes, you need to test that a string is neither null nor empty. Then use

```
if (str != null && str.length() != 0)
```

- You need to test that str is not null first as it is an error to invoke a method on a null value.

String Class: Immutability

- The String class is **immutable**, so that once it is created a String object cannot be changed. If there is a necessity to make a lot of modifications to Strings of characters, then you should use String Buffer & String Builder

```
public class StringDemo {
   public static void main(String args[]) {
      String str = “ABC";
      str = "DEF";
   }
}
```

String Class

- String class has several constructors, but one common way to create a string is by passing a sequence of characters to the constructor.
- new keyword is used to create a new String object, here java does not check for the String Pool and always allocates new memory for a new object in the heap.
- Example:

```
String str1 = "Hello";
String str2 = new String("Hello"); //New Object will be created
//Passing a String Literal as an Argument
String str3 = new String(str2); //3rd Object will be created
//Passing another String Object as an Argument
String str4 = new String(); //A String object referring to a null String.
```


String Class Methods/Functions

- **String concat(String str)**
  - concat ( ) method is used to concatenate one string to the end of another string
  - Example:

```
String s1 = "Indian ";
String s2 = "Cricketer";
String s3 = s1.concat(s2);
System.out.println(s3);                    //IndianCricketer
OR
String s = "Indian" + " Cricketer";
System.out.println(s); //IndianCricketer
```


- **String toLowerCase()**
  - Converts all the characters in the String to lower case.
- **String toUpperCase()**
  - Converts all the characters in the String to upper case.
  - Example:

```
String s = "Indian";
System.out.println(s.toUpperCase());          //INDIAN
System.out.println(s.toLowerCase());          //indian
```


- **String substring (int start)**
- **String substring (int start, int end)**
  - You can extract a substring from given string using substring method.
  - It has two overloaded methods.
  - Example:

```
String s = "IndianCricketer";
System.out.println(s.substring(6));          //Cricketer
System.out.println(s.substring(0,6));        //Indian
```


- **`char charAt(int index)`**
  - The charAt() method returns single character at a specific index in string.
  - Example:

```
String s = "Indian";
System.out.println(s.charAt(0));   //I
System.out.println(s.charAt(3));   //i
```

- **`int length()`**
  - Returns the number of characters in the String.
  - Example:

```
String s = "Indian";
System.out.println(s.length());        //6
```


- **`int indexOf (String s)`**
  - Returns the index within the string of the first occurrence of the specified string.
  - Example:

```
String s = "abaababbaabb";
System.out.println(s.indexOf("aa"));   //2
```

- **`int lastIndexOf(String s)`**
  - Returns the index within the string of the last occurrence of the specified string.
  - Example:

```
String s = "abaababbaabb";
System.out.println(s.lastIndexOf("aa")); //8
```


- **`boolean equals(String another)`**
  - returns true if the strings contain the same characters in the same order, and false otherwise.
  - Remember that this comparison is case-sensitive.
  - Example:

```
String s1 = "Indian";
String s2 = "Indian";
String s3 = "Rahul";
System.out.println(s1.equals(s2));      // true
System.out.println(s1.equals(s3));      // false
```


- **boolean equalsIgnoreCase (String anotherString)**
  - The equalsIgnoreCase( ) performs a comparison of strings that ignores case differences.
  - Example:

```
String s1 = "Indian";
String s2 = "Indian";
String s3 = "INDIAN";
System.out.println(s1.equalsIgnoreCase (s2));  // true
System.out.println(s1.equalsIgnoreCase (s3));  // true
```

- **String split(String regex)**
  - Splits a string into an array of substrings
  - Example:

```
String s6 = new String("abc-def");
for (String retwal : s6.split("-")) {
      System.out.println(retwal);
}// abc
 // def
```


- **String trim()**
  - Returns the copy of the String, by removing whitespaces at both ends, remember that It does not affect whitespaces in the middle.
  - Example:

```
String s1 = " Indian ";
System.out.println(s1.trim( )); // "Indian"
```

- **String replace (char oldChar, char newChar)**
  - Returns new string by replacing all occurrences of oldChar with newChar.
  - Example:

```
String s1 = " Indian ";
System.out.println(s1.replace('d','a'));    // Inaian
```


- **boolean contains(String subString)**
  - The contains() method checks whether a string contains a sequence of characters, it returns true if the characters exist and false if not.
  - Example:

```
String s1 = "Indian";
System.out.println(s1.contains("In"));      // true
System.out.println(s1.contains("Id"));      // false
```

- **String format()**
  - Returns a formatted string using the specified locale, format string, and arguments.
  - Example:

```
String s1 = "Indian";
System.out.println(s1.format("Name is %s", s1);
// Name is Indian
```


- **`int compareTo(String anotherString):`** Compares two string lexicographically.

```
int out = s1.compareTo(s2); //where s1 ans s2 are strings to be compared
```

This returns difference s1-s2.

If :

```
out < 0  // s1 comes before s2
out = 0  // s1 and s2 are equal.
out > 0   // s1 comes after s2.
```

- **`int compareToIgnoreCase( String anotherString):`** Compares two string lexicographically, ignoring case considerations.
  - Calculation is done same as compareTo( ) function above.

StringBuffer

- The StringBuffer class is used to created mutable (modifiable) string, so it is same as String except it is mutable i.e. it can be changed.
- Commonly used Constructors of StringBuffer class:
  - **`StringBuffer( )`**
    - creates an empty string buffer with the initial capacity of 16.
  - **`StringBuffer(int size)`**
    - creates an empty string buffer with the specified capacity as length.
  - **`StringBuffer(String str)`**
    - creates a string buffer with the specified string.
- Example:

```
StringBuffer s = new StringBuffer(“India”);
System.out.println(s.length());  // 5
```


StringBuffer vs. String

String Class	StringBuffer Class
String class object is immutable	StringBuffer class object is mutable
When we create an object of String class by default no additional character memory space is created.	When we create an object of StringBuffer class by default we get 16 additional character memory space.
Less Efficient in terms of memory consumption	Efficient in terms of memory consumption

Math Class

- The **java.lang.Math** class contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

`Math.abs()`	It will return the Absolute value of the given value.
`Math.max()`	It returns the Largest of two values.
`Math.min()`	It is used to return the Smallest of two values.
`Math.round()`	It is used to round of the decimal numbers to the nearest value.
`Math.sqrt()`	It is used to return the square root of a number.
`Math.cbrt()`	It is used to return the cube root of a number.
`Math.pow()`	It returns the value of first argument raised to the power to second argument.

Math Class

`Math.random()`	It returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.
`Math.log()`	It returns the natural logarithm of a double value.
`Math.log10()`	It is used to return the base 10 logarithm of a double value.
`Math.sin()`	It is used to return the trigonometric Sine value of a Given double value.
`Math.cos()`	It is used to return the trigonometric Cosine value of a Given double value.
`Math.sinh()`	It is used to return the trigonometric Hyperbolic Cosine value of a Given double value.
`Math.cosh()`	It is used to return the trigonometric Hyperbolic Sine value of a Given double value.

Math Class

```
public class JavaMathExample1
{
    public static void main(String[] args)
    {
        double x = 28; double y = 4;
        // return the maximum of two numbers
        System.out.println("Maximum number of x and y is: " +Math.max(x, y));
        // return the square root of y
        System.out.println("Square root of y is: " + Math.sqrt(y));
        //returns 28 power of 4 i.e. 28*28*28*28
        System.out.println("Power of x and y is: " + Math.pow(x, y));
          // return the logarithm of given value
       System.out.println("Logarithm of y is: " + Math.log(x));
} }
```

Output:

```
Maximum number of x and y is: 28.0
Square root of y is: 2.0
Power of x and y is: 614656.0
Logarithm of x is: 3.332204510175204
```


Wrapper Classes

- The **Wrapper class in Java** provides the mechanism *to* ***convert primitive to object and object to primitive***.
- Since J2SE 5.0, **autoboxing** and **unboxing** feature convert primitives to objects and objects to primitives automatically. The automatic conversion of primitive into an object is known as autoboxing and vice-versa unboxing.

Wrapper Classes

Primitive Type	Wrapper class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Wrapper Classes: Use

- **Change the value in Method**
- **Serialization**
- **Synchronization**
- **java.util package**
- **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Wrapper Classes: Boxing/Unboxing (Also Auto)

- The conversion from Primitive data type to its object type is known as Boxing and if it is implicitly done by compiler then its known as Auto-boxing.

```
public class WrapperExample1{
  public static void main(String args[]){

      //Converting int into Integer
      int a=20;
      Integer I = Integer.valueOf(a);
      //converting int into Integer explicitly
      Integer j = a;
      //autoboxing, now compiler will write Integer.valueOf(a) internally

      System.out.println(a + " " + i + " " + j);
  }
}
```

Output:

20 20 20

Wrapper Classes: Boxing/Unboxing (Also Auto)

- The conversion from object type to its Primitive data type is known as Unboxing and if it is implicitly done by compiler then its known as Auto unboxing.
- Since Java 5, we do not need to use the xxxValue() method of wrapper classes to convert the wrapper type into primitives.

```
public class WrapperExample2{
    public static void main(String args[]){
        //Converting Integer to int
        Integer a=new Integer(3);

        int i=a.intValue();//converting Integer to int explicitly
        int j=a;
        //unboxing, now compiler will write a.intValue() internally
        System.out.println(a + " " + i + " " + j);
    }
}
```

Output:

3 3 3