1) flow:: JDK >> JRE >> JVM

2) Types of Variables

There are three types of variables in Java:

1. **local variable**: A variable declared inside the body of the method is called local variable.
2. **instance variable**: A variable declared inside the class but outside the body of the method, is called instance variable.
3. **static variable**: A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of static variable and share among all the instances of the class.

```
Eg: class A{
        int data=50;//instance variable
        static int m=100;//static variable
        void method(){
        int n=90;//local variable
        }
}//end of class
```

OOPs (Object-Oriented Programming System):

1. **Object**: An Object can be defined as an instance of a class.
   Eg: Solution defineNewObject = new Solution();
   this defines a new object called "defineNewObject" which acts as an instance of class Solution;

2. **Class**: Collection of objects is called class. It is a logical entity.

3. **Inheritance**: Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.

Eg: class Child extends Parent {

// here Child.java class is extending Parent.java class and can have the control over all the public methods that are defined in Parent.java class.

}

- A class cannot extend multiple classes

Eg: class Child extends Parent1, Parent2 {

Child c = new Child();

c.method();

};

class Parent1 { method(){ } };

class Parent2{ method(){} };

As, both the Parent1 & Parent2 class might contain the same method with same name. By using it inside the Child.java class might create ambiguity with method name.

- So, we come up with the alternative solution called as an interface. Where we define an interface with all the methods inside multiple classes.

Eg: public interface Service {

```
Parent1 method(String abc);

Parent2 method(Integer number);

}
```

and then,

class Child implements Service {

```
Service service = null;

service.method("hello");// this calls Parent1 method();

service.method(123);// this calls Parent2 method();

}
```

4. **Polymorphism :** Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
    a. **Method Overriding:** If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
    b. **Method Overloading:** If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
        We can alter the logic inside the method by overriding the method and the logic inside the child class methods will be executed

    c. **Super keyword:** The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
        i. super can be used to refer immediate parent class instance variable.
        ii. super can be used to invoke immediate parent class method.
        iii. super() can be used to invoke immediate parent class constructor.
    d. **Final Keyword:** The **final keyword** in java is used to restrict the user.
    The java final keyword can be used in many context. Final can be:
        i. **Variable**
        ii. **Method**
        iii. **Class**

5. **Abstraction**

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

6. **Encapsulation:** The **Java Bean** class is the example of a fully encapsulated class.

String

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

- In java, **string objects are immutable**. Immutable simply means unmodifiable or unchangeable. Once string object is created its data or state can't be changed but a new string object is created.

```
class Testimmutablestring{
 public static void main(String args[]){
  String s="Sachin";
  s.concat(" Tendulkar");//concat() method appends the string at the end
  System.out.println(s);//will print Sachin because strings are immutable objects
 }
}
```

- We need to explicitly assign it to the reference variable like
  s = s.concat("Tendulkar");
  System.out.println(s);//will print Sachin Tendulkar

Why string objects are immutable in java?

>> Because java uses the concept of string literal. Suppose there are 5 reference variables, all refers to one object "sachin".If one reference variable changes the value of the object, it will be affected to all the reference variables. That is why string objects are immutable in java.

String Buffer

Java StringBuffer class is used to create mutable (modifiable) string. The StringBuffer class in java is same as String class except it is mutable i.e. it can be changed.

String Builder

Java StringBuilder class is used to create mutable (modifiable) string. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized. It is available since JDK 1.5.

StringBuffer	StringBuilder
StringBuffer is *synchronized* i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is *non-synchronized* i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is *less efficient* than StringBuilder.	StringBuilder is *more efficient* than StringBuffer.