OOPDP

Module I

Syllabus

- **Module I**
  :Introduction to Classes, Objects and Java

- Introduction to Object Technology, Java, Understanding the Java development environment, Programming in Java, Memory concepts, Doing basic Arithmetic, Comparing entities, Classes, Objects, Methods, Strings, Primitive vs reference types.

Introduction to Object Technology

- As the name suggests, Object-Oriented Programming or OOPs refers to languages that uses objects in programming.
- The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

- **OOPs Concepts:**
- Class
- Object
- Polymorphism
- Inheritance
- Encapsulation
- Abstraction

- **Class:**
- A class is a user defined blueprint or prototype from which objects are created.
- It represents the set of properties or methods that are common to all objects of one type.

- **Object:**
- An object is an instance of a class
- An object consists of:
  - **State** : It is represented by attributes of an object.
  - **Behavior** : It is represented by methods of an object
  - **Identity** : It gives a unique name to an object and enables one object to interact with other objects.

Example1:

Class/Object- Dogs

State/Attributes- size, age, color, breed, etc.

Methods/Behavior- eat, sleep, sit and run.

Example 2:

Class/Object: House

State: Address, Color, Area

Behavior: Open door, close door

- **Polymorphism:**
- Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently.
- Polymorphism in Java are mainly of 2 types:
  1. Overloading in Java
  2. Overriding in Java

- Overloading in Java
  - Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both.
  - Also a compile-time (or static) polymorphism.
- Overriding in Java
  - Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
  - When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.
  - It is an example of Run time polymorphism in Java

- **Inheritance:**
- Inheritance is an important pillar of OOP(Object Oriented Programming).
- It is the mechanism in java by which one class is allow to inherit the features(fields and methods) of another class.
  - The class whose features are inherited is known as **superclass(or a base class or a parent class).**
  - The class that inherits the other class is known as **subclass(or a derived class, extended class, or child class)**.
  - The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Advantage:** Inheritance supports the concept of "reusability".
- In Java the keyword used for inheritance is **extends**.

- **Encapsulation:**
- It is defined as the wrapping up of data under a single unit.
- It is the mechanism that binds together code and the data it manipulates.
- Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.

- **Data Abstraction:**
- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.
- Ex: A car is viewed as a car rather than its individual components.
- In java, abstraction is achieved by **interfaces** and **abstract classes**.

Java

- **Java** is an object-oriented programming language developed by James Gosling and colleagues at Sun Microsystems in the early 1990s.
- The language itself borrows much syntax from C and C++ but has a simpler object model and fewer low-level facilities.
- **Java is not** fully **object oriented** because it supports primitive data type like int,byte,long etc.,which are **not objects**.

Features of Java

- Simple
- Object-Oriented
- Portable
- Platform independent
- Secured
- Robust
- Architecture neutral
- Interpreted
- High Performance
- Multithreaded
- Distributed
- Dynamic

- **Simple**
  - Java syntax is based on C++ (so easier for programmers to learn it after C++).
  - Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
  - There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.
- **Object-oriented**
  - Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.
- **Platform Independent**
  - Java code is compiled by the compiler and converted into bytecode.
  - This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

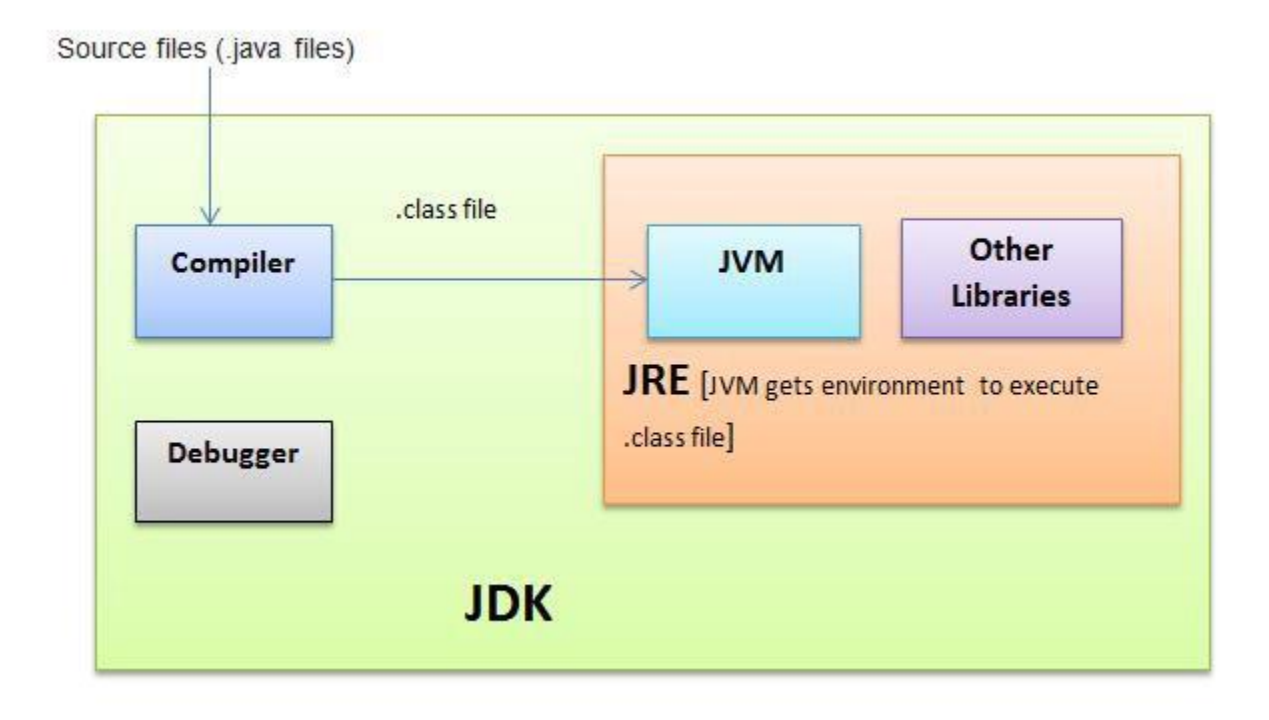
- **Secured**
  - **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically.
  - It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
  - **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
  - **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.
- **Robust**
  - Robust simply means strong. Java is robust because:
  - It uses strong memory management.
  - There is a lack of pointers that avoids security problems.
  - There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
  - There are exception handling and the type checking mechanism in Java. All these points make Java robust.

- **Architecture-neutral**
  - Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.
- **Portable**
  - Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.
- **Distributed**
  - Java is distributed because it facilitates users to create distributed applications in Java.
  - RMI and EJB are used for creating distributed applications.
- **Multi-threaded**
  - A thread is like a separate program, executing concurrently.
  - We can write Java programs that deal with many tasks at once by defining multiple threads.
- **Dynamic**
  - Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand.
  - Java supports dynamic compilation and automatic memory management (garbage collection).

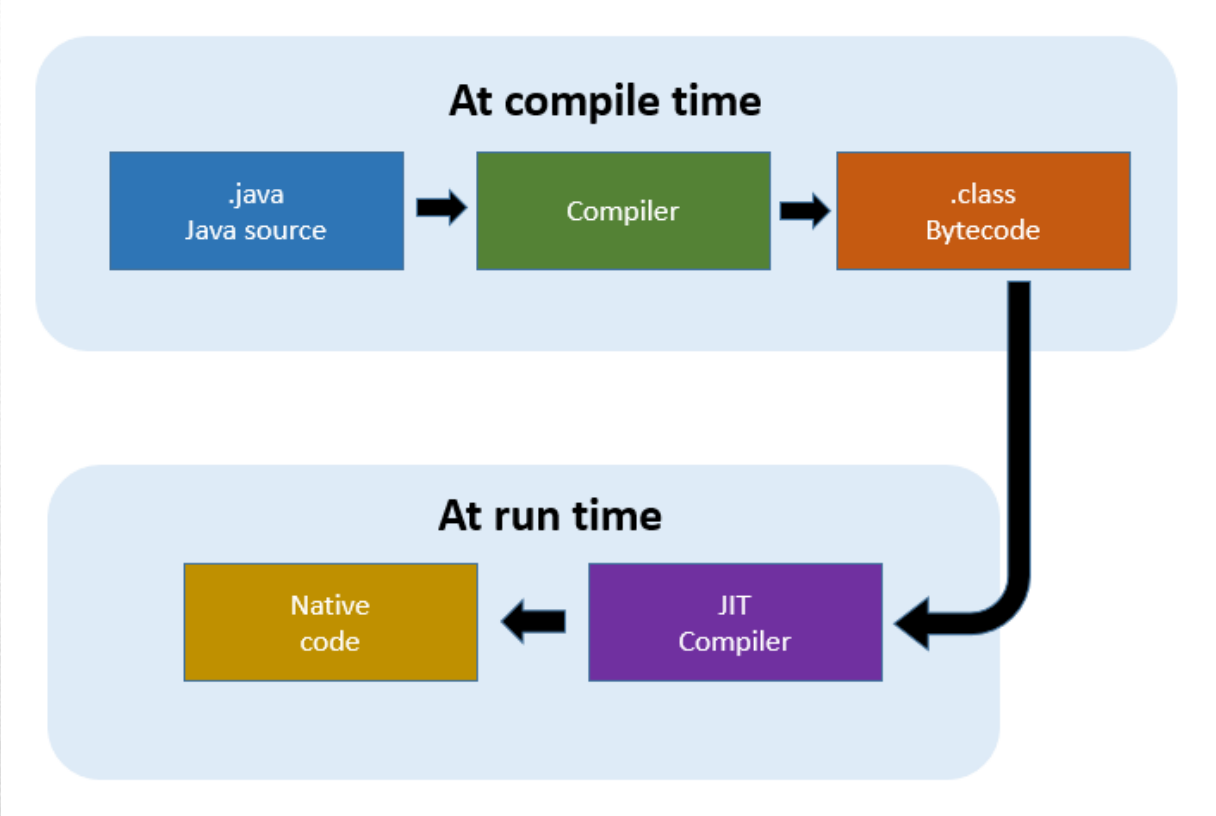
Java development environment

- **JDK**(Java Development Kit) :
  - JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.
  - It contains JRE + development resources.
- **JRE**(Java Runtime Environment) :
  - JRE contains the parts of the Java libraries required to run Java programs.
  - JRE can be view as a subset of JDK.
- **JVM**(Java Virtual Machine):
  - JVM is an abstract machine.
  - It is a specification that provides runtime environment in which java bytecode can be executed.

- The JVM does the following primary tasks:
  - Loads code
  - Verifies code
  - Executes code
  - Provides runtime environment


Source files (.java files)
.class file
Compiler
JVM
Other Libraries
JRE [JVM gets environment to execute .class file]
Debugger
JDK

- **Java JIT(Just In Time) Compiler:**
  - JIT compilers interact with the Java Virtual Machine (JVM) at run time and compile suitable bytecode sequences into native machine code.
  - While using a JIT compiler, the hardware is able to execute the native code, as compared to having the JVM interpret the same sequence of bytecode repeatedly and incurring an overhead for the translation process.
  - This subsequently leads to performance gains in the execution speed, unless the compiled methods are executed less frequently.


At compile time
.java
Java source
Compiler
.class
Bytecode
At run time
Native
code
JIT
Compiler

Programming in Java

- A *class* is a collection of *fields* (data) and *methods* (procedure or function) that operate on that data.
- The basic syntax for a class definition:

[Access Specifier] class *ClassName* [extends *SuperClassName*]

{

[fields declaration]

[methods declaration]

}

Example:

```
public class Circle {

    public double x, y; // centre of the circle
    public double r;    // radius of circle

    //Methods to return circumference and area
    public double circumference() {
            return 2*3.14*r;
    }
    public double area() {
            return 3.14 * r * r;
   }
}
```

```
class MyMain
{
	public static void main(String args[])
	{
		Circle c1;  // creating reference
		c1 = new Circle(); // creating object
		c1.x = 10;  // assigning value to data field
		c1.y = 20;
		c1.r = 5;
		double area = c1..area(); // invoking method
		double circumf = c1.circumference();
		System.out.println(" Area="+area);
		System.out.println(" Circumference ="+circumf);
	}
}
```

- **Output:**

Area=78.5

Circumference =31.400000000000002

- In Java, a constructor is a block of codes similar to the method.
- It is called when an instance of the class is created.
- At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It is called constructor because it constructs the values at the time of object creation.
- It is not necessary to write a constructor for a class.
- It is because java compiler creates a default constructor if your class doesn't have any.

Memory Concepts

- **Rules for creating Java constructor**
  - Constructor name must be the same as its class name
  - A Constructor must have no explicit return type
  - A Java constructor cannot be abstract, static, final, and synchronized
- **There are two types of constructors in Java:**
  1. Default constructor (no-arg constructor)
  2. Parameterized constructor
- Note: The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

```
class Student3{
 int id;
String name;
 void display(){
System.out.println(id+" "+name);
}
public static void main(String args[]){
 Student3 s1=new Student3(); //default constructor called
 Student3 s2=new Student3();   //default constructor called
s1.display();
s2.display();
}
 }
```

Output:

```
0 null
0 null
```

Default Cosatructor

```
class Student{
 int id;
String name;      //creating a parameterized constructor
Student(int i,String n){
  id = i;
name = n;
}      //method to display the values
void display(){
System.out.println(id+" "+name);
}
public static void main(String args[]){
Student s1 = new Student(111,"Karan");
Student s2 = new Student(222,"Aryan");
s1.display();
s2.display();     }  }
```

Output:

```
111 Karan
222 Aryan
```

Static variables

- When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.
- **Important points for static variables :-**
- We can create static variables at class-level only.
- static block and static variables are executed in order they are present in a program.

```
class Student{
  int rollno;
  String name;
static String college ="ITS";//static variable
Student(int r, String n)
{
rollno = r;
   name = n;
 }     //method to display the values
   void display ()
{
System.out.println(rollno+" "+name+" "+college);
}  }
public class TestStaticVariable1
{
public static void main(String args[])
{
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
s1.display();
s2.display();   }  }
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

Static Methods

- When a method is declared with *static* keyword, it is known as static method.
- The most common example of a static method is *main( )* method.
- Any static member can be accessed before any objects of its class are created, and without reference to any object.
- Methods declared as static have several restrictions:
  - They can only directly call other static methods.
  - They can only directly access static data.
  - They cannot refer to this or super in any way.

```
class Student
{
   int rollno;
    String name;
  static String college = "ITS";
     static void change()
{
college = "BBDIT";
   }
  Student(int r, String n)
{       rollno = r;
   name = n;
 }       //method to display values
void display(){
System.out.println(rollno+" "+name+" "+college);
}  }  //Test class to create and display the values of object
public class TestStaticMethod{
 public static void main(String args[]){
 Student.change(); //use class name to call since it is static method
 Student s1 = new Student(111,"Karan");
Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
s1.display();
 s2.display();
  s3.display();
    }  }
```

Outptut:

```
111 Karan BBDIT
222 Aryan BBDIT
333 Sonoo BBDIT
```

- **Types of variables**
  - Local Variables
  - Instance Variables
  - Static Variables

- **Local Variables**:
- A variable defined within a block or method or constructor is called local variable.
- These variable are created when the block in entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access these variable only within that block.
- Initilisation of Local Variable is Mandatory.

- **Instance Variables**:
- Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- Initilisation of Instance Variable is not Mandatory. Its default value is 0
- Instance Variable can be accessed only by creating objects.

- **Static Variables**:
- Static variables are also known as Class variables.
- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initilisation of Static Variable is not Mandatory. Its default value is 0
- If we access the static variable like Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name to class name automatically.

- **Reference Variables:**
- Reference variables are not pointers but a handle to the object which is created in heap memory
- **Primitive Vs Reference Variables**
- The main difference between primitive and reference type is that *primitive type always has a value*, it can never be null but reference type can be null, which denotes the absence of value.
- So if you create a primitive variable of type **int** and forget to initialize it then it's value would be 0, the default value of integral type in Java, but a reference variable by default has a null value, which means no reference is assigned to it.

- When you assign a value to primitive data types, the primitive value is copied, but when you assign an object to reference type, the handle is copied. which means for reference type object is not copied only the handle is copied, i.e. the object is shared between two reference variable
- primitive variables using equality (==) operator, their primitive values are compared but when you compare reference variable, their address is compared, which means two objects which are logically equal e.g. two String object with same content may be seen as not equal,

Strings in Java

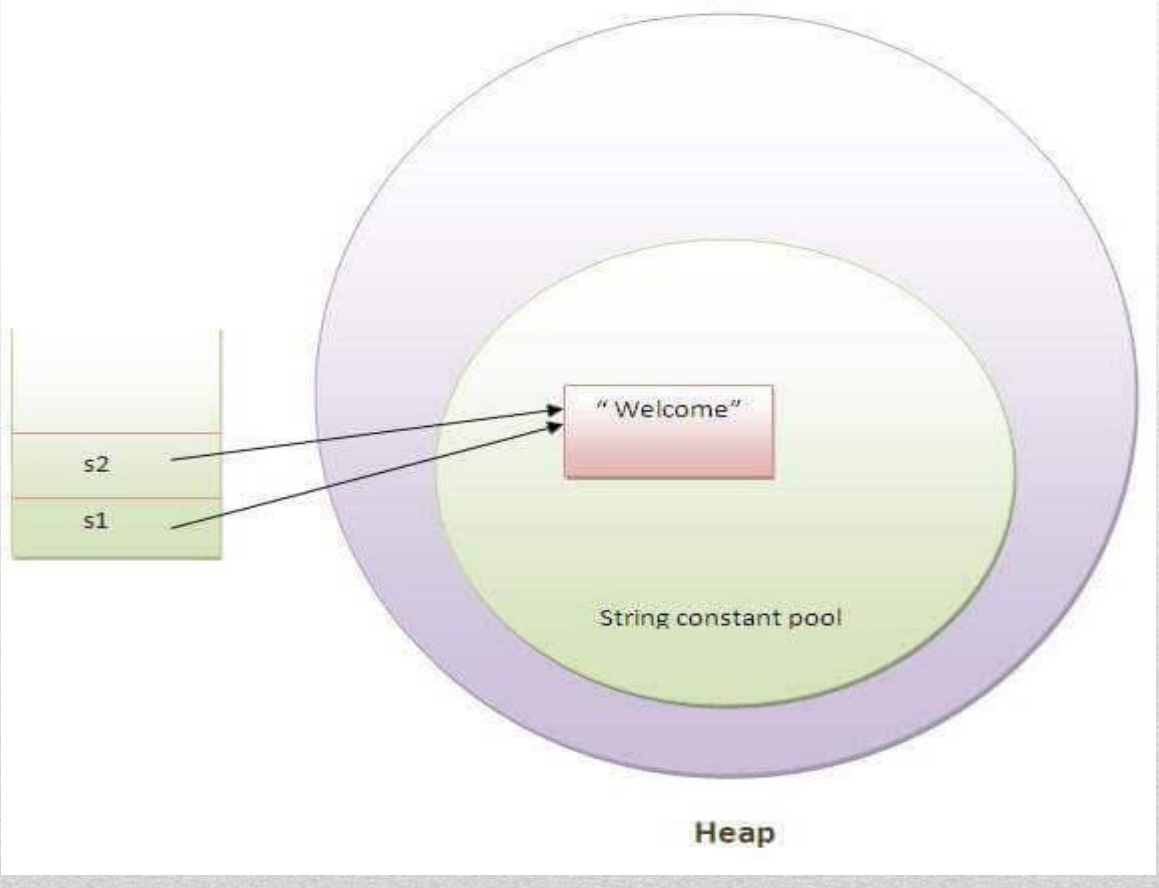
- Generally, String is a sequence of characters.
- But in Java, string is an object that represents a sequence of characters.
- The java.lang.String class is used to create a string object.

- There are two ways to create String object:

1. By string literal
2. By new keyword

- Java String literal is created by using double quotes.
- String s="welcome";
- Each time you create a string literal, the JVM checks the "string constant pool" first.
- If the string already exists in the pool, a reference to the pooled instance is returned.
- If the string doesn't exist in the pool, a new string instance is created and placed in the pool.

- String s1="Welcome";
- String s2="Welcome";//It doesn't create a new instance
- It makes Java more memory efficient (because no new objects are created if it exists already in the string constant pool).



- String s=**new** String("Welcome");//creates two objects an d one reference variable
- In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool.
- The variable s will refer to the object in a heap (non-pool).

```
public class Main
        {
public static void main(String[] args)
        {
String s1=new String("hai");
String s2="hai";
String s3="hai";
System.out.println(s1==s2);
System.out.println(s3==s2);

        }}
```

Examples:

- **Output:**

false

true

Immutability

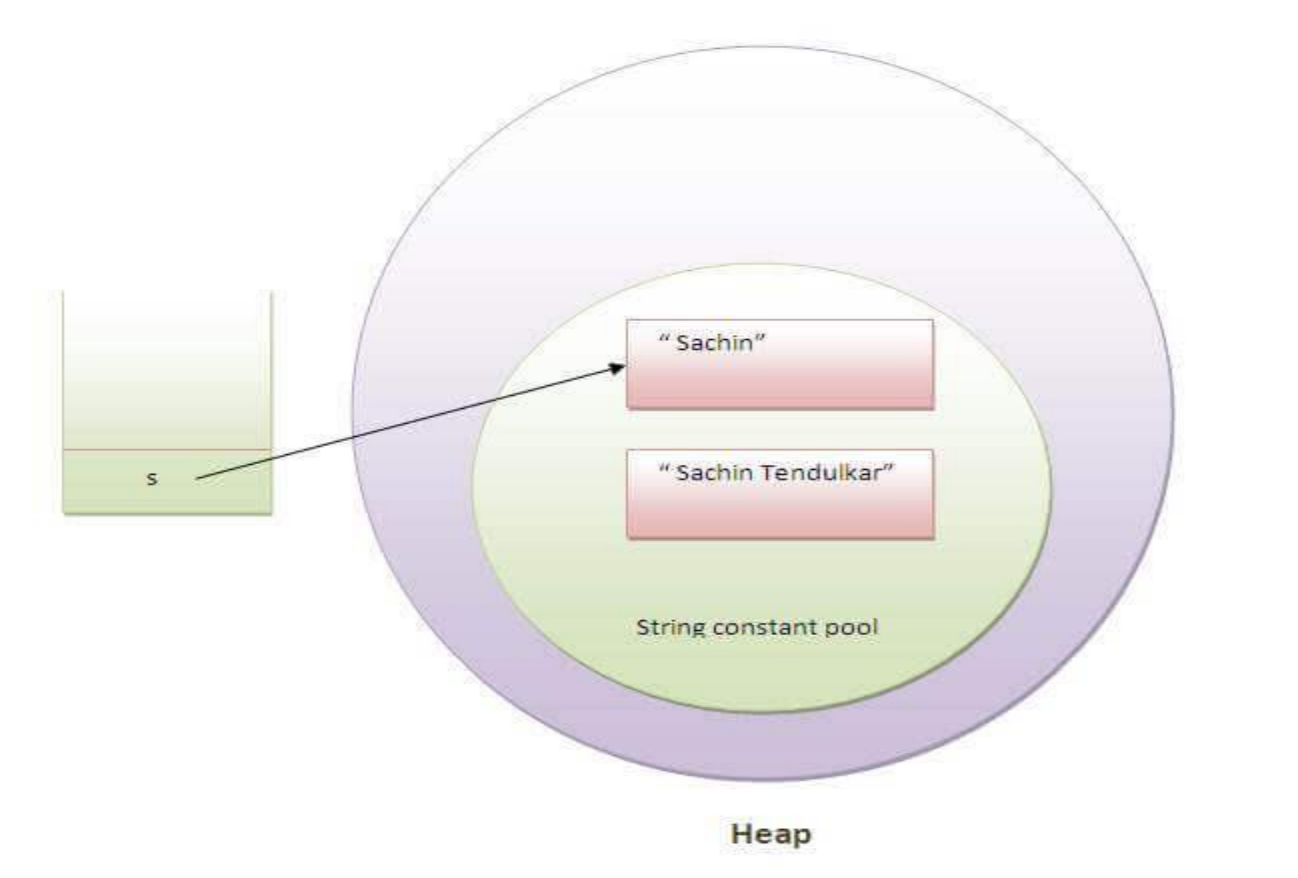
- In java, **string objects are immutable**.
- Immutable simply means unmodifiable or unchangeable.
- Once string object is created its data or state can't be changed but a new string object is created.

- String pool is possible only because String is immutable in Java.
- If String is not immutable then it would cause a severe security threat to the application.
- Since String is immutable, it is safe for multithreading

```
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
 }
}
```

- **Output:**

Sachin



```
class Testimmutablestring1{
 public static void main(String args[]){
   String s="Sachin";
   s=s.concat(" Tendulkar");
   System.out.println(s);
 }
}
```

`

- **Output:**

Sachin Tendulkar

Note:Still ‘Sachin’ object is not modified.

String Comparison

- There are three ways to compare string in java:
- By equals() method
  - The String equals() method compares the original content of the string.
- By = = operator
  - The = = operator compares references not values.
- By compareTo() method
  - The String compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.
  - Suppose s1 and s2 are two string variables. If:
    - **s1 == s2** :0
    - **s1 > s2** :positive value
    - **s1 < s2** :negative value

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   String s4="Saurav";
   System.out.println(s1.equals(s2));
   System.out.println(s1.equals(s3));
   System.out.println(s1.equals(s4));
 }
}
```

- **Output:**

true

true

false

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);
   System.out.println(s1==s3);
}
```

- **Output:**

true

false

```
class Teststringcomparison4{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));
   System.out.println(s1.compareTo(s3));
   System.out.println(s3.compareTo(s1));
 }
}
```

- **Output:**

0

1

-1

```
public class HelloWorld
{     public static void main(String []args)
{
String s1="Sachin";
String s2="Sachin";
String s3="Uat";
System.out.println(s1.compareTo(s2));
System.out.println(s1.compareTo(s3));
System.out.println(s3.compareTo(s1));
      }}
```

- **Output:**

0

-2

2

Java String class methods

- The java.lang.String class provides a lot of methods to work on string.
- By the help of these methods, we can perform operations on string such as trimming, concatenating, converting, comparing, replacing strings etc.

```
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
```

- **Output:**

```
SACHIN
sachin
Sachin
```

toUpperCase()
toLowerCase()

String s="Sachin";

System.out.println(s.charAt(0));//S

System.out.println(s.charAt(3));//h

- **Output:**

S

h

charAt()

String s="Sachin";

System.out.println(s.length());//6

- Output:

6

length()

```
int a=10;
String s=String.valueOf(a);  //converts the given type into String
System.out.println(s+10);
```

- **Output:**

1010

valueOf()

```
String s1="Java is a programming language. Java is a platfo
    rm. Java is an Island.";
String replaceString=s1.replace("Java","Kava");
//replaces all occurrences of "Java" to "Kava"
System.out.println(replaceString);
```

Output:

Kava is a programming language. Kava is a platform. Kava is an Island.

replace()

```
String x = "0123456789"; // the value of each char is the same as its index!
System.out.println( x.substring(5) ); // output is "56789“
System.out.println( x.substring(5, 8)); // output is "567"
```

substring()

String s = “Java”;

Char [] arrayChar = s.toCharArray(); **//this will produce array of size 4**

toCharArray()

contains()

```
String x = “Java is programming language”;
System.out.println(x.contains(“Amit”)); // This will print false
System.out.println(x.contains(“Java”)); // This will print true
```

StringBuffer class

- **StringBuffer** may have characters and substrings inserted in the middle or appended to the end.
- It will automatically grow to make room for such additions and often has more characters preallocated than are actually needed, to allow room for growth.
- **Java StringBuffer class is thread-safe i.e. multiple threads cannot access it simultaneously. So it is safe and will result in an order.**

Constructors

- **StringBuffer()** : creates an empty string buffer with the initial capacity of 16.
- **StringBuffer(String str)**: creates a string buffer with the specified string.
- **StringBuffer(int capacity)**: creates an empty string buffer with the specified capacity as length.

```
class StringBufferExample{
 public static void main(String args[]){
 StringBuffer sb=new StringBuffer("Hello ");
sb.append("Java"); //now original string is changed
System.out.println(sb);//prints Hello Java
 }  }
```

StringBuffer Methods

append()

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```

insert()

```
class StringBufferExample3{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.replace(1,3,"Java");
System.out.println(sb);//prints HJavalo
}
}
```

replace()

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```

reverse()

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity()); //default 16
sb.append("Hello");
System.out.println(sb.capacity()); //now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2

}
}
```

capacity()

StringBuilder class

- 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.

Constructors

- **StringBuilder()** :creates an empty string Builder with the initial capacity of 16.
- **StringBuilder(String str)** :creates a string Builder with the specified string.
- **StringBuilder(int length)** :creates an empty string Builder with the specified capacity as length.

toString Method

- Every class in java is child of Object class either directly or indirectly.
- Object class contains toString() method.
- We can use toString() method to get string representation of an object.
- Whenever we try to print the Object reference then internally toString() method is invoked.
- If we did not define toString() method in your class then Object class toString() method is invoked otherwise our implemented/Overridden toString() method will be called.

```
class A{
int a;
}
public class Mainclass{
   public static void main(String []args){
A ob=new A();
A ob1=new A();
System.out.println(ob);
System.out.println(ob1);
}}
```

Output:

```
A@6d06d69c
A@7852e922
```

```
class A{
 int a;
public String toString()
{       return("A Object");
 }
 }
public class Mainclass{
  public static void main(String []args)
{
A ob=new A();
A ob1=new A();
System.out.println(ob);
System.out.println(ob1);
   }}
```

- **Output:**

A Object

A Object