UNIT 3

INHERITANCE AND INTERFACES

INHERITANCE

Java, Inheritance is an important pillar of OOP(Object-Oriented Programming). It is the mechanism in Java by which one class is allowed to inherit the features(fields and methods) of another class. In Java, Inheritance means creating new classes based on existing ones. A class that inherits from another class can reuse the methods and fields of that class. In addition, you can add new fields and methods to your current class as well.

Need of Inheritance

- Code Reusability: The code written in the Superclass is common to all subclasses. Child classes can directly use the parent class code.
- Method Overriding: Method Overriding is achievable only through Inheritance. It is one of the ways by which Java achieves Run Time Polymorphism.

Terminologies

- Class: Class is a set of objects which shares common characteristics/ behavior and common properties/ attributes. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
- Super Class/Parent Class/Base Class: The class whose features are inherited is known as a superclass(or a base class or a parent class).
- Sub Class/Child Class/Derived Class: The class that inherits the other class is known as a 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.
- Reusability: Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Extends keyword

The extends keyword indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

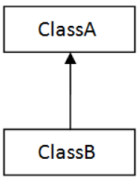
Syntax :

```
class DerivedClass extends BaseClass
{
   //methods and fields
}
```

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance
4. Multiple Inheritance
5. Hybrid Inheritance

Single Inheritance



1) Single

In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes, it is also known as simple inheritance. In the below figure, 'A' is a parent class and 'B' is a child class. The class 'B' inherits all the properties of the class 'A'.

Program:

```
class Animal
{
        void eat()
        {
                System.out.println("Eating");

        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Barking");

        }
        public static void main(String[] args)
        {
                Dog d = new Dog();
                d.eat();
                d.bark();
        }
}
```

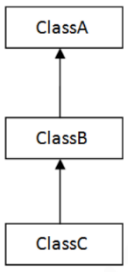
```
E:\JAVA\ANU>javac Dog.java

E:\JAVA\ANU>java Dog
Eating
Barking
```

Multilevel Inheritance

In Multilevel Inheritance, a derived class will be inheriting a base class, and as well as the derived class also acts as the base class for other classes. In the below image, class A serves as a base class for the derived class B, which in turn serves as a base class for the derived class C.

Program:

```
class Animal
{
        void eat()
        {
                System.out.println("Eating");

        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Barking");

        }
}
class BabyDog extends Dog
{
        void sleep()
        {
                System.out.println("Sleeping");

        }
        public static void main(String[] args)
        {
                BabyDog b = new BabyDog();
                b.eat();
                b.bark();
                b.sleep();
        }
}
```



2) Multilevel

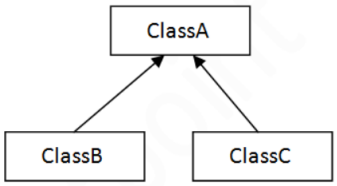
```
E:\JAVA\ANU>javac BabyDog.java

E:\JAVA\ANU>java BabyDog
Eating
Barking
Sleeping
```

Hierarchical Inheritance

When two or more classes inherits a single class, or one class serves as a superclass (base class) for more than one subclass. In the below image, class A serves as a base class for the derived classes B, C, and D.

Program:

```
class Animal
{
        void eat()
        {
                System.out.println("Eating");

        }
}
class Dog extends Animal
{
        void bark()
        {
                System.out.println("Barking");

        }
}
class Cat extends Animal
{
        void meow()
        {
                System.out.println("Meowing");

        }
        public static void main(String[] args)
        {
                Cat c = new Cat();
                c.eat();
                //c.bark();  ERROR
                c.meow();
        }
}
```



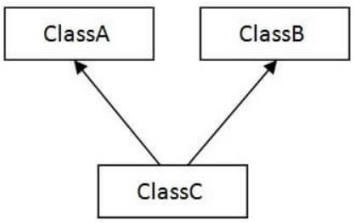
3) Hierarchical

```
E:\JAVA\ANU>javac Cat.java

E:\JAVA\ANU>java Cat
Eating
Meowing
```

Multiple Inheritance (Through Interfaces)

In Multiple inheritances, one class can have more than one superclass and inherit features from all parent classes. Please note that Java does not support multiple inheritances with classes. In Java, we can achieve multiple inheritances only through Interfaces.

```
class A
{
        void msg1()
        {
                System.out.println("Hello");
        }
}
class B
{
        void msg2()
        {
                System.out.println("Welcome");
        }
}
class C extends A,B //suppose if it were
{

        public static void main(String args[])
        {
                C obj=new C();
                obj.msg1();
                obj.msg2();
        }
}
```

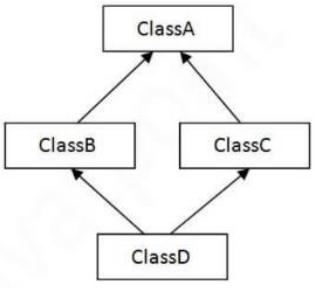


4) Multiple

```
Compile Time Error
```

Hybrid Inheritance

It is a mix of two or more of the above types of inheritance. Since Java doesn't support multiple inheritances with classes, hybrid inheritance involving multiple inheritance is also not possible with classes. In Java, we can achieve hybrid inheritance only through Interfaces.



5) Hybrid

Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Method overloading in Java is also known as Compile-time Polymorphism, or Static Polymorphism or Early binding.
- It is similar to constructor ovaloading in java (that allows a class having More than one constructor having different parameter List.

Different ways to overload the method

There are two ways to overload the method in java

- ✓ By changing number of arguments
  - int add(int a,int b)
  - int add(int a,int b,int c)
- ✓ By changing the data type
  - void add(int a, float b)

Program:

```
class OverLoad
{
        void add(int a,int b)
        {
                System.out.println(a+b);
        }
        void add(float a,float b)
        {
                System.out.println(a+b);
        }
        void add(int a,int b,int c)
        {
                System.out.println(a+b+c);
        }
        public static void main(String args[])
        {
                OverLoad ol=new OverLoad();
                ol.add(10,20);
                ol.add(2.3f,4.4f);
                ol.add(4,2,6);
        }
}
```

```
E:\JAVA\ANU>javac OverLoad.java

E:\JAVA\ANU>java OverLoad
30
6.7
12
```

Method Overriding

- Method Overriding is achieved through Inheritance.
- In Java, Overriding is a feature that allows a subclass is having a method which is already present in its super-classes or parent classes.
- Declaring a method in subclass which is already present in parent class is known as Method overiding .
- Overriding is done so that A Child class can give its own implementation to a method which is already provided by parent class. In this case the method in parent class is called Overridden method and the method in child class is called overriding Method.
- Method Overriding in java is also known as Runtime Polymorphism or Dynamic Polymorphism or Late Binding.
- Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

Rules:-

- Method must have same name in the Parent class.
- Method must have same parameters as in the parent class method.
- The return type must be same as the Return type declared in original Overridden method in the super class
- We cannot override a method marked with final and you cannot override a method marked with Static.

Program:

```
class Parent
{
        void msg()
        {
                System.out.println("parent Overridden method");
        }
}
class OverRide extends Parent
{
        void msg()
        {
                System.out.println("child Overriding method ");
        }
        public static void main(String args[])
        {
                OverRide or=new OverRide();
                or.msg();
        }
}
```

```
E:\JAVA\ANU>javac OverRide.java

E:\JAVA\ANU>java OverRide
child Overriding method
```

Difference Between Compile-time polymorphism and Runtime polymorphism

Method Overloading	Method Overriding
Method overloading is a compile-time polymorphism.	Method overriding is a run-time polymorphism.
Method overloading helps to increase the readability of the program.	Method overriding is used to grant the specific implementation of the method which is already provided by its parent class or superclass.
It occurs within the class.	It is performed in two classes with inheritance relationships.
Method overloading may or may not require inheritance.	Method overriding always needs inheritance.
In method overloading, methods must have the same name and different signatures.	In method overriding, methods must have the same name and same signature.
In method overloading, the return type can or can not be the same, but we just have to change the parameter.	In method overriding, the return type must be the same or co-variant.

Method Overloading	Method Overriding
Static binding is being used for overloaded methods.	Dynamic binding is being used for overriding methods.
Private and final methods can be overloaded.	Private and final methods can't be overridden.
The argument list should be different while doing method overloading.	The argument list should be the same in method overriding.

Super Keyword in Java

- The super keyword in Java is a reference variable which is used to refer immediate parent class object.
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class methods.
3. super() can be used to invoke immediate parent class constructor.

➢ **super can be used to invoke immediate parent class method mainly used to call in method overriding to call the parent class method.**

When you override a method in a subclass, you may still want to call the original method from the superclass to maintain some of its behavior. You can achieve this using the super keyword. This is particularly useful if the superclass method has some useful functionality that you want to extend or modify in the subclass.

```
class Parent
{
        void msg()
        {
                System.out.println("Parent Method");
        }
}
class SuperMeth extends Parent
{
        void msg()
        {
                super.msg();
                System.out.println("Child Method");
        }
        public static void main(String args[])
        {
                SuperMeth sm=new SuperMeth();
                sm.msg();
        }
}
```

```
E:\JAVA\ANU>javac SuperMeth.java

E:\JAVA\ANU>java SuperMeth
Parent Method
Child Method
```

Final keyword

The final keyword in java is used to restrict the user. It is used to indicate that a variable, method, or class cannot be modified or extended. The java final keyword can be used in many context. Final can be used with:

- ✓ final variables – to create constant variables
- ✓ final methods – to prevent Method Overriding

- ✓ final classes – to prevent Inheritance

➢ **Final with variables**: When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.

```
class FinalVar
{
        public static void main(String args[])
        {
                final float pi=3.14f;
                //pi=3.146798f; ERROR
                System.out.println("final variable "+pi);
        }
}
```

```
E:\JAVA\ANU>javac FinalVar.java

E:\JAVA\ANU>java FinalVar
final variable 3.14
```

➢ **Final with methods**: When a method is declared as final, it cannot be overridden by a subclass. This is useful for methods that are part of a class's public API and should not be modified by subclasses.

```
class Parent
{
       final void msg()
       {
              System.out.println("parent");
       }
}
class FinalMeth extends Parent
{
       void msg()
       {
              System.out.println("child");
       }
       public static void main(String args[])
       {
              FinalMeth fm=new FinalMeth();
              fm.msg();
       }
}
```

```
E:\JAVA\ANU>javac FinalMeth.java
FinalMeth.java:10: error: msg() in FinalMeth cannot override msg()
 in Parent
        void msg()
             ^
  overridden method is final
1 error
```

➢ **Final with classes**: When a class is declared as final, it cannot be extended by a subclass. This is useful for classes that are intended to be used as is and should not be modified or extended.

```
final class Parent
{
       int a=10;
}
class FinalClass extends Parent
{
       public static void main(String args[])
       {
              FinalClass c=new FinalClass();
              System.out.println(c.a);
       }
}
```

```
E:\JAVA\ANU>javac FinalClass.java
FinalClass.java:5: error: cannot inherit from final Parent
class FinalClass extends Parent
                         ^
1 error
```

Polymorphism

- Polymorphism in Java is a concept by which we can perform a single action in different ways. Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in Java
  - ✓ compile-time polymorphism (or) early binding (or) method overloading
  - ✓ runtime polymorphism (or) late binding (or) method overriding
- We can perform polymorphism in java by method overloading and method overriding. (**for method overloading and overriding - refer previous topics**)

Abstract clases

Before learning the Java abstract class, let's understand the abstraction in Java first.

- **Abstraction** is a process of hiding the implementation details and showing only functionality to the user.

- Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
- Abstraction lets you focus on what the object does instead of how it does it.

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

- ✓ Abstract class (0 to 100%)
- ✓ Interface (100%)

abstract class

- A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).
- Points to Remember
  1. An abstract class must be declared with an abstract keyword.
  2. It can have abstract and non-abstract methods.
  3. We can have an abstract class without any abstract method.
  4. If a class contains at least one abstract method then compulsory should declare a class as abstract
  5. It cannot be instantiated (or) An instance of an abstract class can not be created.
  6. An abstract class can have data members, abstract methods, non-abstract methods, constructors and main() method.
  7. There can be a final method in abstract class but any abstract method in abstract class can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: "Illegal combination of modifiers: abstract and final".
  8. If the Child class is unable to provide implementation to all abstract methods of the Parent class then we should declare that Child class as abstract so that the next level Child class should provide implementation to the remaining abstract method

Ex:

```
abstract class A
{
        //abstract methods
        //Non abstract methods
}
```

abstract Method

- A method declared using the abstract keyword within an abstract class and does not have a definition (implementation) is called an abstract method.
- When we need just the method declaration in a super class, it can be achieved by declaring the methods as abstracts.
- Abstract method is also called subclass responsibility as it doesn't have the implementation in the super class. Therefore a subclass must override it to provide the method definition.

  Syntax `abstract return_type method_name( [ argument-list ] );`

- Points to Remember
  1. An abstract method do not have a body (implementation), they just have a method signature (declaration). The class which extends the abstract class implements the abstract methods.

2. If a non-abstract (concrete) class extends an abstract class, then the class must implement all the abstract methods of that abstract class. If not the concrete class has to be declared as abstract as well.
3. As the abstract methods just have the signature, it needs to have semicolon (;) at the end.

Program

```
abstract class Animal
{
        abstract void sound();
}
class Dogs extends Animal
{
        void sound()
        {
                System.out.println("Barking");
        }
        public static void main(String args[])
        {
                Dogs d=new Dogs();
                d.sound();
        }
}
```

```
E:\JAVA\ANU>javac Dogs.java

E:\JAVA\ANU>java Dogs
Barking
```

Java program on static fields and Methods

- **Static** variables and methods belong to a class and are called with the Class Name rather than using object variables, like ClassName.methodName();
- There is only one copy of a static variable or method for the whole class. For example, the main method is static because there should only be 1 main method.
- Static methods only have access to other static variables and static methods. Static methods cannot access or change the values of instance variables or the this reference (since there is no calling object for them), and static methods cannot call non-static methods. However, non-static methods have access to all variables (instance or static) and methods (static or non-static) in the class.

```
class StaticFields
{
        static int a=100;
        static void sum(int a,int b)
        {
                System.out.println("Static method Addition "+(a+b));
        }
        public static void main(String[] args)
        {
                StaticFields.sum(10,5);
                System.out.println("Static variable "+StaticFields.a);
        }
}
```

```
E:\JAVA\ANU>javac StaticFields.java

E:\JAVA\ANU>java StaticFields
Static method Addition 15
Static variable 100
```

Interfaces

- Interfaces is similar to a class (or) it looks like a class but it is not a class.
- An interface can have methods & variables just like a class. but the methodis declared in interface are by default public and abstract (only method signature no body). also the variables declared in interface are public, Static & final by default.

Uses:

- ✓ It is used to achieve abstraction. Since Methods in interface donot have any body, they have to be implemented by class before you can access them.
- ✓ By interface we can support functionality of multiple inheritance

Syntax:

```
interface InterfaceName
{
        //variables
        //methods;
}
```

Ex: After

```
interface MyInterface
{
        int x=10;
        void method1();
}
```

Comilation

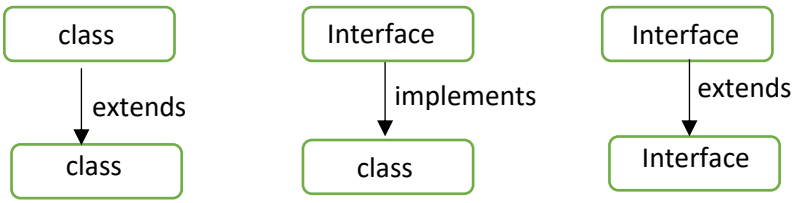
```
interface MyInterface
{
        public static final int x=10;
        public abstract void method1();
}
```

Example program

```
interface MyInterface
{
        void method1();
        void method2();
}
class InterDemo implements MyInterface
{
        public void method1()
        {
                System.out.println("Method1 Implementation");
        }
        public void method2()
        {
                System.out.println("Method2 Implementation");
        }
        public static void main(String args[])
        {
                InterDemo id=new InterDemo();
                id.method1();
                id.method2();
        }
}
```

```
E:\JAVA\ANU>javac InterDemo.java

E:\JAVA\ANU>java InterDemo
Method1 Implementation
Method2 Implementation
```

Interface and inheritance



Program on inheritance with interfaces:

```
interface Inf1
{
        void method1();
}
interface Inf2 extends Inf1
{
        void method2();
}
class Demo implements Inf2
{
        public void method1()
        {
                System.out.println("Inf1 Implementation");
        }
        public void method2()
        {
                System.out.println("Inf2 Implementation");
        }
        public static void main(String args[])
        {
                Demo d=new Demo();
                d.method1();
                d.method2();
        }
}
```

```
E:\JAVA\ANU>javac Demo.java

E:\JAVA\ANU>java Demo
Inf1 Implementation
Inf2 Implementation
```

- In the above program an interface can not implement another interface . It has to extend the other interface. Here we have two interfaces Inf1 and Inf2.
- Inf2 extends Inf1 so the class implemets the Inf2 it has to provide implementation of all the methods of interface Inf2 as well as Inf1.

Keypoints

Here are the some keypoints to remember about the interfaces

- We can not instantiate an Interface in Java (can't creates any object in interface).
- Interface provides full abstraction as non of its methods have body. In Other hand abstract class provides partial abstraction (0% to 100%) & interface provides full abstraction (100%).
- An Interface can Extend any interface but cannot implement. Class implements interface and interface Extends interface.
- implements keyword is used by class to implement an interface.
- A Class can implements any number of interfaces (Multiple inheritance can be achieve by interfaces).
- All the interface Method are by default Abstract & public.
- While providing implementation in class of any method of an interface it needs to be mentioned as public.
- Variable declared in interface are public, State & final by default.

Ex

```
interface Inter
{
        public static final float PI = 3.14;
}
```

- Interface Variables must be initialized at the time of declaration otherwise Compiles will raises an error. If there are two or more same methodes in two interfaces and a class implements both interfaces, implementation of method Once is enough.

  Program for Multiple Inheritance

```
interface A
{
	void add(int a,int b);
}
interface B
{
	void add(int a,int b);
}
class SameMeth implements A,B
{
	public void add(int a,int b)
	{
		System.out.println("Addition ="+(a+b));

	}
	public static void main(String args[])
	{
		SameMeth sm=new SameMeth();
		sm.add(10,20);
	}
}
```

```
E:\JAVA\ANU>javac SameMeth.java

E:\JAVA\ANU>java SameMeth
Addition =30
```

- Variable name conflits are resolved by interface name.

```
interface A
{
	int a=10;
}
interface B
{
	int a=20;
}
class SameVar implements A,B
{
	static int a=30;
	public static void main(String args[])
	{
		System.out.println("A Variable "+A.a);
		System.out.println("B Variable "+B.a);
		System.out.println("Class Variable "+SameVar.a);
	}
}
```

```
E:\JAVA\ANU>javac SameVar.java

E:\JAVA\ANU>java SameVar
A Variable 10
B Variable 20
Class Variable 30
```

Nesting of methods

In Java, methods and variables that are defined within a class can only be accessed by creating an instance of that class or by using the class name if the methods are static. The dot operator is used to access methods and variables within a class. However, there is an exception to this rule: a method can also be called by another method using the class name, but only if both methods are present in the same class. Efficient code organization and simplified method calls within a class are possible through nesting of methods. Understanding how methods can call other methods within the same class is an important aspect of Java programming.

```
** SYNTAX **
class Nesting
{
    method1()
    {
        // statements
    }

    method2()
    {
      // calling method1() from method2()
       method1();
    }
    method3()
    {
      // calling of method2() from method3()
      method2();
    }
}
```

```
interface A
{
	void add(int a,int b);
}
interface B
{
	void add(int a,int b);
}
class SameMeth implements A,B
{
	public void add(int a,int b)
	{
		System.out.println("Addition ="+(a+b));

	}
	public static void main(String args[])
	{
		SameMeth sm=new SameMeth();
		sm.add(10,20);
	}
}
```

```
E:\JAVA\ANU>javac NestingMethod.java

E:\JAVA\ANU>java NestingMethod
average is 22
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface those are as:

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();
	}

String class

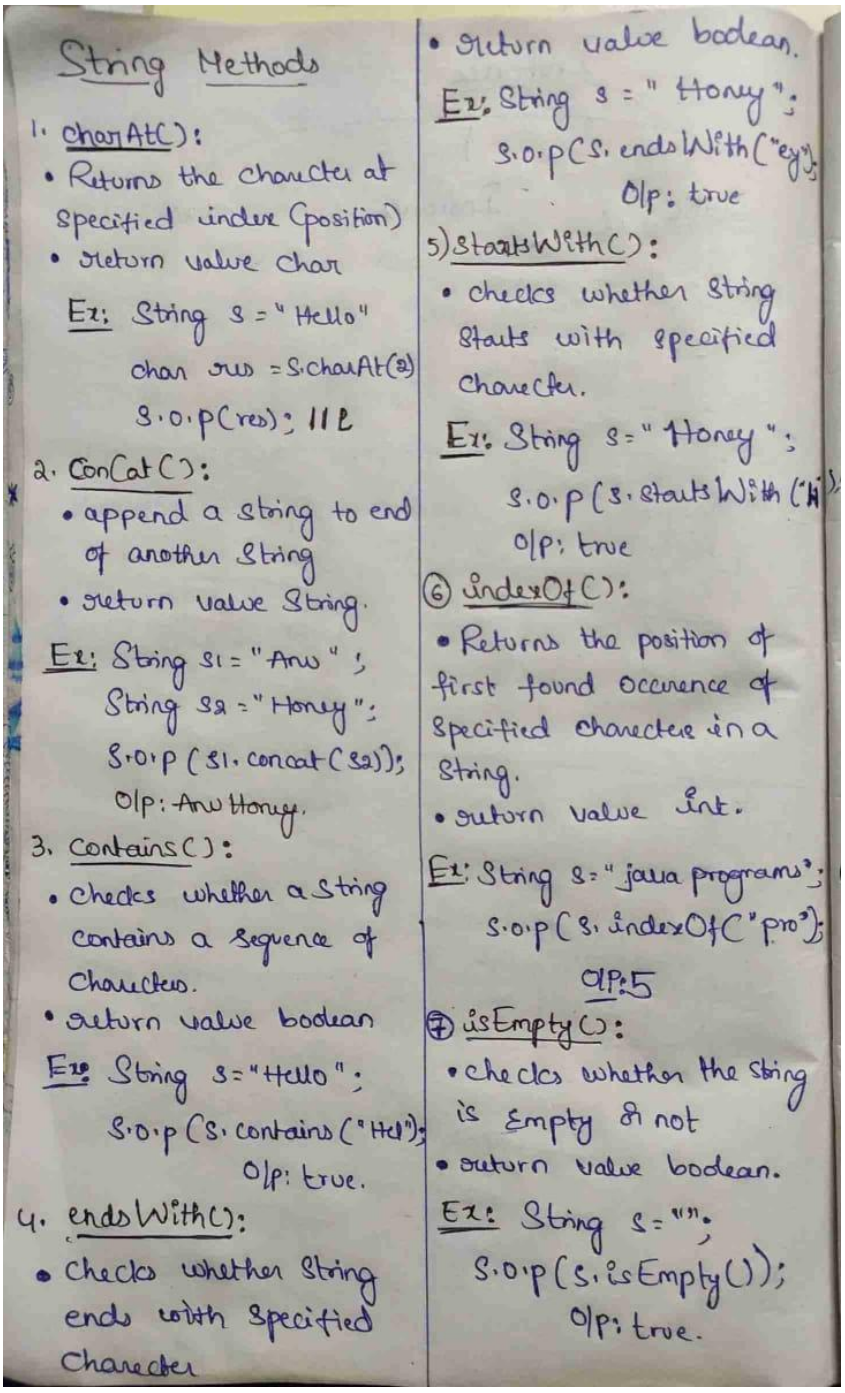
- In the Java programming language, strings are objects.
- The Java platform provides the String class to create and manipulate strings.
- String is basically an object that represents sequence of characters. An array of characters works same as Java string.
- We can create Strings in 2 ways:

```
String s = "Hello";
String str = new String("Hello");
```

- We can also create String by using character Array

```
char[] arr = { 'h', 'e', 'l', 'l', 'o', '.' };
String str = new String(arr);
System.out.println(str);
```

String class methods



String Methods

1. charAt():
   - Returns the character at specified index (position)
   - return value char

   Ex: String s = "Hello"
   char res = s.charAt(2)
   s.o.p(res); // l

2. ConCat():
   - append a string to end of another String
   - return value String.

   Ex: String s1 = "Anu";
   String s2 = "Honey";
   s.o.p(s1.concat(s2));
   o/p: Anu Honey.

3. contains():
   - Checks whether a string contains a sequence of characters.
   - return value boolean

   Ex: String s = "Hello";
   s.o.p(s.contains("Hel"));
   o/p: true.

4. endsWith():
   - Checks whether String ends with specified character
   - return value boolean.

   Ex: String s = "Honey";
   s.o.p(s.endsWith("ey"));
   o/p: true

5) startsWith():
   - checks whether String starts with specified character.

   Ex: String s = "Honey";
   s.o.p(s.startsWith("H"));
   o/p: true

6) indexOf():
   - Returns the position of first found occurence of specified characters in a String.
   - return value int.

   Ex: String s = "java programs";
   s.o.p(s.indexOf("pro"));
   o/p: 5

7) isEmpty():
   - checks whether the string is Empty or not
   - return value boolean.

   Ex: String s = "";
   s.o.p(s.isEmpty());
   o/p: true.

⑧ join():

- joins one & more Strings with a specified separator.
- returns String.

Ex:

```
String str = String.join("-",
  "Apple", "Ball", "cat");
S.O.p(str);
```

O/P: Apple-Ball-cat.

⑨ length():

- Returns length of specified String.
- return value int

Eg: String s = "programmer";

```
S.O.p(s.length());
```

O/P: 10

⑩ split():

- splits a string into array of substrings
- returns String.

Eg:

```
String s = "this is split fun";
String arr[] = s.split(" ");
for (String i: arr)
    S.O.p(s)
```

O/P: this
is
split
fun

String COMPARISION

- ==
- equals()
- compareTo()

• (==)

```
String s = "Hel";
S.O.p(s == "Hel") //true
```

• equals(): (case sensitive)

- compares 2 strings returns
  - → true if equal
  - → false if not-equal

Eg: String s = "Hi";

```
S.O.p(s.equals("Hi");
                //true
```

• equalsIgnoreCase():
case Insensitive.

• CompareTo():

- compares 2 strings lexicographically.

It returns:

- 0 → if Both s1 == s2
- +ve → if s1 > s2
- -ve → if s2 < s2

Eg:

```
String s1 = "Hella";
String s2 = "Hello";
S.O.p(s1.compareTo(s2));
        O/P: -14
S.O.p(s2.compareTo(s1));
        O/P: 14
```

• CompareToIgnoreCase():
Same but case Insensitive

Java program on searching a string in an Array of Strings

```
class StringSearch
{
        public static void main(String args[])
        {
                String s[]={"e","u","a","o","i"};
                String key="o";
                boolean found=false;
                for(int i=0;i<s.length;i++)
                {
                        if(key.equals(s[i]))
                        {
                                found=true;
                                break;
                        }
                }
                if(found)
                        System.out.println("FOUND");
                else
                        System.out.println("NOT FOUND");
        }
}
```

```
E:\JAVA\ANU>javac StringSearch.java

E:\JAVA\ANU>java StringSearch
FOUND
```

Java Program on String Sorting

```
import java.util.Arrays;
class StringSort
{
        public static void main(String args[])
        {
                String s[]={"e","u","a","o","i"};
                System.out.println("Before Sorting "+Arrays.toString(s));
                Arrays.sort(s);
                System.out.print("After Sorting   ");
                boolean found=false;
                for(String i: s)
                {
                        System.out.print(i+" ");
                }

        }
}
```

```
E:\JAVA\ANU>javac StringSort.java

E:\JAVA\ANU>java StringSort
Before Sorting [e, u, a, o, i]
After Sorting   a e i o u
```