CA 105(A)
Java Programming (Core Java)

UNIT-II Introduction to OOPs:

OOPs concepts

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. **Object-Oriented Programming** is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Predefined classes(String, StringBuffer)

String Class:-

What is String 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.

How to create a string object?

There are two ways to create String object:

1) By string literal

2) By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

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. For example:

String s1="Welcome";

String s2="Welcome";//It doesn't create a new instance

2) By new keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

Java String Example:-

StringExample.java

```
public class StringExample{
public static void main(String args[]){
String s1="java";//creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}}
```

Output:

```
java
strings
example
```

• Java String class methods:-

- The java.lang.String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index) Eg:- String name=“imrd"; **char** ch=name.charAt(3); ans:-d	It returns char value for the particular index
2	int length() Eg:- String name=“imrd"; **char** ch=name.length(); ans:-4	It returns string length
5	String substring(int beginIndex) String s1=“RCPIMRD"; System.out.println(s1.substring(2)); O/P:- PIMRD	It returns substring for given begin index.
6	String substring(int beginIndex, int endIndex) String s1=“RCPIMRD"; System.out.println(s1.substring(2,4)); O/P:- PI	It returns substring for given begin index and end index.

10	boolean equals(Object another)
	public class EqualsExample{
	public static void main(String args[]){
	String s1="java";
	String s2="java";
	String s3="JAVA";
	String s4="python";
	System.out.println(s1.equals(s2));//true because content and case is same
	System.out.println(s1.equals(s3));//false because case is not same
	System.out.println(s1.equals(s4));//false because content is not same
	}}
		It checks the equality of string with the given object.

11	boolean isEmpty()	It checks if string is empty.

12	String concat(String str)
	public class ConcatExample2 {
	public static void main(String[] args) {
	String str1 = "Hello";
	String str2 = "Java";
	String str3 = "Reader";
	// Concatenating one string
	String str4 = str1.concat(str2);
	System.out.println(str4);
	// Concatenating multiple strings
	String str5 = str1.concat(str2).concat(str3);
	System.out.println(str5);
	}
	}
		It concatenates the specified string.

13	String replace(char old, char new)	It replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	It replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another) **Input :** str1 = "pAwAn"; str2 = "PAWan" str2.equalsIgnoreCase(str1); **Output :**true	It compares another string. It doesn't check case.
19	int indexOf(int ch) Eg:- String s1="imrd"; **int** i=s1.indexOf('r');	It returns the specified char value index.

23	String toLowerCase() String s1=“RCPIMRD"; String s2=s1.toLowerCase();	It returns a string in lowercase.
25	String toUpperCase()	It returns a string in uppercase.

27	String trim()
	public class StringTrimExample{
	public static void main(String args[]){
	String s1=" hello string ";
	System.out.println(s1+"java");//without trim()
	System.out.println(s1.trim()+"java");//with trim()
	}}
	Output
	hello string java
	hello stringjava
		It removes beginning and ending spaces of this string.

28	static String valueOf(int value)	It converts given type into string. It is an overloaded method.

StringBuffer class in Java

StringBuffer is a class in Java that represents a mutable sequence of characters. It provides an alternative to the immutable String class, allowing you to modify the contents of a string without creating a new object every time.

Here are some important features and methods of the StringBuffer class:

1. StringBuffer objects are mutable, meaning that you can change the contents of the buffer without creating a new object.
2. The initial capacity of a StringBuffer can be specified when it is created, or it can be set later with the ensureCapacity() method.
3. The append() method is used to add characters, strings, or other objects to the end of the buffer.
4. The insert() method is used to insert characters, strings, or other objects at a specified position in the buffer.
5. The delete() method is used to remove characters from the buffer.
6. The reverse() method is used to reverse the order of the characters in the buffer.

- **Example of using StringBuffer to concatenate strings:**

```
public class StringBufferExample {
        public static void main(String[] args)
        {
                StringBuffer sb = new StringBuffer();
                sb.append("Hello");
                sb.append(" ");
                sb.append("world");
                String message = sb.toString();
                System.out.println(message);
        }
}
```

- **Output**
- Hello world
- **1. append() method**
- The append() method concatenates the given argument with this string.
- **Example:**

```
import java.io.*;

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

- **Output:-** Hello Java

- **2. insert() method**
- The insert() method inserts the given string with this string at the given position.
- **Example:**

```
import java.io.*;
class A {
        public static void main(String args[])
        {
                StringBuffer sb = new StringBuffer("Hello ");
                sb.insert(1, "Java");
                // Now original string is changed
                System.out.println(sb);
        } }
```

- **Output:-** HJavaello
- **3. replace() method**
- The replace() method replaces the given string from the specified beginIndex and endIndex-1.
- **Example:**

```
import java.io.*;
class A {
        public static void main(String args[])
        {
                StringBuffer sb = new StringBuffer("Hello");
                sb.replace(1, 3, "Java");
                System.out.println(sb);
        }  }
```

- **Output:-** HJavalo

- **4. delete() method**
- The delete() method of the StringBuffer class deletes the string from the specified beginIndex to endIndex-1.
- **Example:**
- import java.io.*;

- class A {
- public static void main(String args[])
- {
- StringBuffer sb = new StringBuffer("Hello");
- sb.delete(1, 3);
- System.out.println(sb);
- }
- }
- **Output:-** Hlo
- **5. reverse() method:-** The reverse() method of the StringBuilder class reverses the current string.
- **Example:**
- import java.io.* ;
- class A {
- public static void main(String args[])
- {
- StringBuffer sb = new StringBuffer("Hello");
- sb.reverse();
- System.out.println(sb);
- }
- } **Output:-** olleH

Methods of Java StringBuffer class

Methods	Action Performed
append()	Used to add text at the end of the existing text.
length()	The length of a StringBuffer can be found by the length( ) method
capacity()	the total allocated capacity can be found by the capacity( ) method
charAt()	This method returns the char value in this sequence at the specified index.
delete()	Deletes a sequence of characters from the invoking object
deleteCharAt()	Deletes the character at the index specified by the *loc*
ensureCapacity ()	Ensures capacity is at least equal to the given minimum.
insert()	Inserts text at the specified index position
length()	Returns the length of the string
reverse()	Reverse the characters within a StringBuffer object
replace()	Replace one set of characters with another set inside a StringBuffer object

Type Casting in Java

In Java, **type casting** is a method or process that converts a data type into another data type in both ways **manually and automatically**.

The **automatic conversion is done** by the compiler and manual conversion performed by the programmer.

Type casting:-

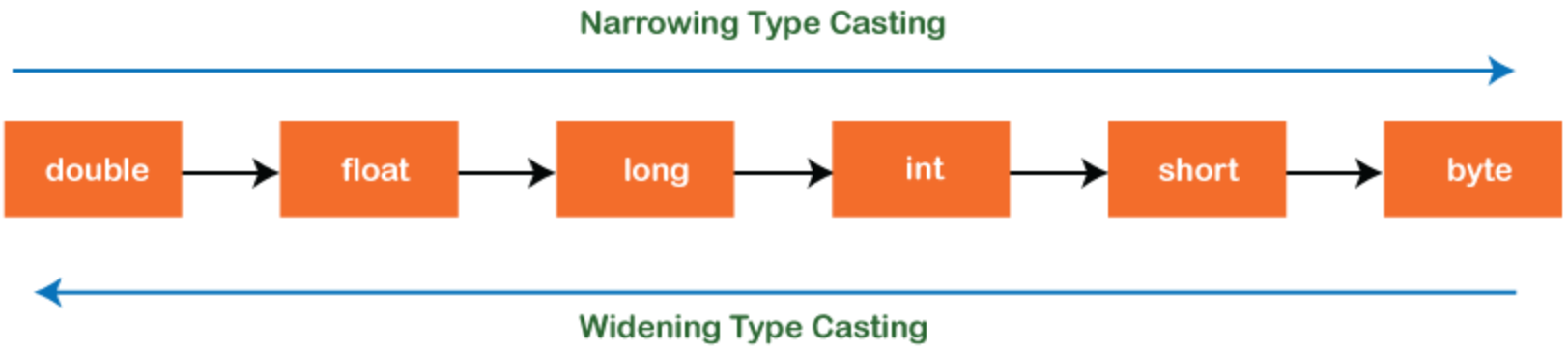
Convert a value from one data type to another data type is known as **type casting**.

Types of Type Casting:-

There are **two** types of type casting:

1) Widening Type Casting (automatically)

2) Narrowing Type Casting (manually)



Type Casting in Java

Type Casting in Java

- 1) Widening Type Casting:-
- Converting a lower data type into a higher one is called **widening** type casting.
- It is also known as **implicit conversion** or **casting down**.
- It is done **automatically**. It is safe because there is no chance to lose data.
- Both data types must be compatible with each other.
- The target type must be larger than the source type.
- **byte** -> **short** -> **char** -> **int** -> **long** -> **float** -> **double**
- For example:-

```
public class WideningTypeCastingExample
{
public static void main(String[] args)
{
int x = 7;
//automatically converts the integer type into long type
long y = x;
//automatically converts the long type into float type
float z = y;
System.out.println("Before conversion, int value "+x);
System.out.println("After conversion, long value "+y);
System.out.println("After conversion, float value "+z);
} }
```

Output

```
Before conversion, the value is: 7
After conversion, the long value is: 7
After conversion, the float value is: 7.0
```

Type Casting in Java

- 2) Narrowing Type Casting :-
- Converting a higher data type into a lower one is called **narrowing** type casting.
- It is also known as **explicit conversion** or **casting up**.
- It is done **manually** by the programmer.
- If we do not perform casting then the compiler reports a compile-time error.
- **double** -> **float** -> **long** -> **int** -> **char** -> **short** -> **byte**
- For example:-

```
public class NarrowingTypeCastingExample
{   public static void main(String args[])
{
double d = 166.66;
//converting double data type into long data type
long l = (long)d;
//converting long data type into int data type
int i = (int)l;
System.out.println("Before conversion: "+d);
//fractional part lost
System.out.println("After conversion into long type: "+l);
//fractional part lost
System.out.println("After conversion into int type: "+i);
} }
```

Output

Before conversion: 166.66 After conversion into long type: 166 After conversion into int type: 166

Wrapper classes Java OR Java Autoboxing and Unboxing

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

- **Use of Wrapper classes in Java:-**
- Java is an object-oriented programming language, so we need to deal with objects many times like in Collections, Serialization, Synchronization, etc. Let us see the different scenarios, where we need to use the wrapper classes.

1. **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
2. **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
3. **Synchronization:** Java synchronization works with objects in Multithreading.
4. **java.util package:** The java.util package provides the utility classes to deal with objects.
5. **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 Java OR Java Autoboxing and Unboxing

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

➢ Autoboxing:-

- ➢ The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing,
- ➢ for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.
- ➢ We do not need to use the valueOf() method of wrapper classes to convert the primitive into objects.
- ➢ **Wrapper class Example: Primitive to Wrapper**

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
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
```

➢ Unboxing:-

- ➢ The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing.
- ➢ It is the reverse process of autoboxing.
- ➢ we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

- **Wrapper class Example: Wrapper to Primitive**

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
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

- **Java Basic Input and Output**
- **Java Output**
- In Java, you can simply use
- System.out.println(); or
- System.out.print(); or
- System.out.printf();
- to send output to **standard output** (screen).
- Here,
- System is a class
- out is a public static field: it accepts output data.
- **Difference between println(), print() and printf()**
- **print() -** It prints string inside the quotes.
- **println() -** It prints string inside the quotes similar like print() method. Then the cursor moves to the beginning of the next line.
- **printf() -** It provides string formatting (similar to printf in C/C++ programming).
- **Java Input**
- Java provides different ways to get input from the user. you will learn to get input from user using the object of **Scanner** class.
- In order to use the object of Scanner, we need to **import java.util.Scanner** package.

- Then, we need to create an object of the Scanner class.
- We can use the object to take input from the user.
- // create an object of Scanner
- Scanner input = new Scanner(System.in);
- // take input from the user
- int number = input.nextInt();
- **Example: Get Integer Input From the User**

```
import java.util.Scanner;
class Input {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int number = input.nextInt();
        System.out.println("You entered " + number);
        // closing the scanner object
        input.close();
    } }
```

Output:

```
Enter an integer: 23
You entered 23
```

- *In the above example, we have created an object named input of the Scanner class.*
- *We then call the nextInt() method of the Scanner class to get an integer input from the user.*
- *Similarly, we can use*

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

User defined class

- A class is a template or blueprint that is used to create objects.
- Class representation of objects and the sets of operations that can be applied to such objects.
- A class consists of Data members and methods.
- The primary purpose of a class is to hold data/information. This is achieved with attributes which are also known as data members.
- The member functions determine the behavior of the class, i.e. provide a definition for supporting various operations on data held in the form of an object.

- Defining a Class in Java
- **Syntax:**
- public class class_name
- {
- Data Members;
- Methods;
- }

Example:

```
public class Car
{
   public:
   double color; // Color of a car
   double model; // Model of a car
}
```

- Private, Protected, Public is called visibility labels.
- The members that are declared private can be accessed only from within the class.
- Public members can be accessed from outside the class also.

- Class Members
- *Data* and *functions* are members.
- Data Members and methods must be declared within the class definition
- Example:

```
//ODD EVEN
import java.util.Scanner;
 class CheckEvenOdd
{
  public static void main(String args[])
  {
    int num;
    System.out.println("Enter an Integer number:");
     //The input provided by user is stored in num
    Scanner input = new Scanner(System.in);
    num = input.nextInt();
     /* If number is divisible by 2 then it's an even number
     * else odd number*/
    if ( num % 2 == 0 )
        System.out.println("Entered number is even");
     else
        System.out.println("Entered number is odd");
  }
}
```

Access Modifiers in Java

- There are four types of Java access modifiers:-

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

finalize() Method

- Finalize() is the method of Object class.
- This method is called just before an object is garbage collected.
- finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

- **Syntax:-**
- **protected void** finalize() **throws** Throwable
- Example 1

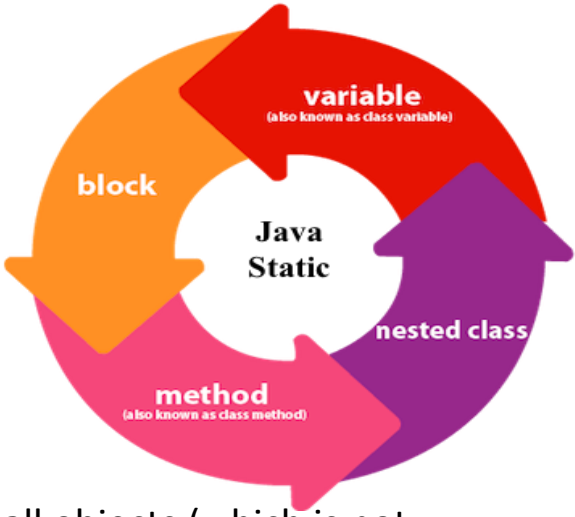
```
public class JavafinalizeExample1 {
    public static void main(String[] args)
    {
        JavafinalizeExample1 obj = new JavafinalizeExample1();
        System.out.println(obj.hashCode());
        obj = null;
        // calling garbage collector
        System.gc();
        System.out.println("end of garbage collection");
    }
    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    } }
```

Output:

2018699554
end of garbage collection
finalize method called

static keyword

- The **static keyword** in Java is used for memory management mainly.
- We can apply static keyword with variables, methods, blocks and nested classes.
- The static keyword belongs to the class than an instance of the class.
- A static variable is **common to all the instances** (or objects) of the class because it is a class level variable.
- In other words you can say that only a **single copy of static variable** is created and shared among all the instances of the class.
- Memory allocation for such variables only happens once when the class is loaded in the memory.



- **The static can be:**

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) Java static variable

- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.

- **Advantages of static variable**

- It makes your program **memory efficient** (i.e., it saves memory).

Example:- Program of counter by static variable.

Static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

```
//Java Program to illustrate the use of static variable which
//is shared with all objects.
class Counter2{
static int count=0;//will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}
  public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Output:

```
1
2
3
```

2) Java static method:-

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

- **Example:-**
- //Java Program to get the cube of a given number using the static method
-
- **class** Calculate{
- **static int** cube(**int** x){
- **return** x*x*x;
- }
-
- **public static void** main(String args[]){
- **int** result=Calculate.cube(5);
- System.out.println(result);
- }
- }

Output:125

- Restrictions for the static method
- There are two main restrictions for the static method. They are:

1. The static method can not use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

3) Java static block:-

- If you need to do computation in order to initialize your **static variables**, you can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.

1. Is used to initialize the static data member.
2. It is executed before the main method at the time of classloading.

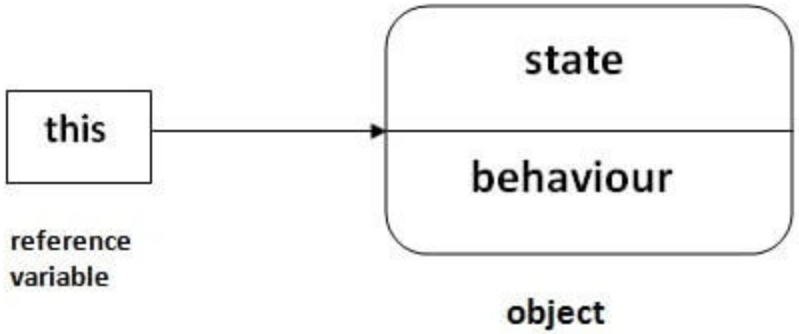
```
// Java program to demonstrate use of static blocks
classTest
{
    // static variable
    staticinta = 10;
    staticintb;

    // static block
    static{
        System.out.println("Static block initialized.");
        b = a * 4;
    }

    publicstaticvoidmain(String[] args)
    {
       System.out.println("from main");
       System.out.println("Value of a : "+a);
       System.out.println("Value of b : "+b);
    }
}
```

Output:

```
Static block initialized.
from main
Value of a : 10
Value of b : 40
```

this keyword

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



- **Usage of Java this keyword**
- Here is given the 6 usage of java this keyword.

1. this can be used to refer current class instance variable.
2. this can be used to invoke current class method (implicitly)
3. this() can be used to invoke current class constructor.
4. this can be passed as an argument in the method call.
5. this can be passed as argument in the constructor call.
6. this can be used to return the current class instance from the method.

1) this: to refer current class instance variable

The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
1. class Student{
2. int rollno;
3. String name;
4. float fee;
5. Student(int rollno,String name,float fee){
6. rollno=rollno;
7. name=name;
8. fee=fee;
9. }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12. class TestThis1{
13. public static void main(String args[]){
14. Student s1=new Student(111,"ankit",5000f);
15. Student s2=new Student(112,"sumit",6000f);
16. s1.display();
17. s2.display();
18. }}
```

- **Output:**
- 0 null 0.0
- 0 null 0.0

In the above example, parameters (formal arguments) and instance variables are same. So, we are using this keyword to distinguish local variable and instance variable.

Solution of the above problem by this keyword

```
1.    class Student{
2.  int rollno;
3.  String name;
4.  float fee;
5.  Student(int rollno,String name,float fee){
6.  this.rollno=rollno;
7.  this.name=name;
8.  this.fee=fee;
9.  }
10. void display(){System.out.println(rollno+" "+name+" "+fee);}
11. }
12.
13. class TestThis2{
14. public static void main(String args[]){
15. Student s1=new Student(111,"ankit",5000f);
16. Student s2=new Student(112,"sumit",6000f);
17. s1.display();
18. s2.display();
19. }}
```

Output:

```
111   ankit     5000.0
112   sumit     6000.0
```

2) this: to invoke current class method

You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method. Let's see the example

```
1.  class A{
2.  void m(){System.out.println("hello m");}
3.  void n(){
4.  System.out.println("hello n");
5.  //m();//same as this.m()
6.  this.m();
7.  }
8.  }
9.  class TestThis4{
10. public static void main(String args[]){
11. A a=new A();
12. a.n();
13. }}
```

Output:

hello n

hello m

Inner class

- Inner class means one class which is a member of another class. There are basically four types of inner classes in java.
- 1) Nested Inner class
  2) Method Local inner classes
  3) Anonymous inner classes
  4) Static nested classes
- **1) Nested Inner class** can access any private instance variable of outer class. Like any other instance variable, we can have access modifier private, protected, public and default modifier. Like class, interface can also be nested and can have access specifiers.
- ***Following example demonstrates a nested class.***

```
class Outer {
  // Simple nested inner class
  class Inner {
    public void show() {
        System.out.println("In a nested class method");
    } } }
class Main {
  public static void main(String[] args) {
    Outer.Inner in = new Outer().new Inner();
    in.show();
  }
}
```

Output:
In a nested class method

Inner class

- **2) Method Local inner classes**
  Inner class can be declared within a method of an outer class. In the following example, Inner is an inner class in outerMethod().

```
class Outer {
   void outerMethod() {
      System.out.println("inside outerMethod");
      // Inner class is local to outerMethod()
      class Inner {
         void innerMethod() {
            System.out.println("inside innerMethod");
         }
      }
      Inner y = new Inner();
      y.innerMethod();
   }
}
class MethodDemo {
   public static void main(String[] args) {
      Outer x = new Outer();
      x.outerMethod();
   }
}
```

Output
Inside outerMethod
Inside innerMethod

Inner class

- **3) Static nested classes**
  Static nested classes are not technically an inner class. They are like a static member of outer class.

```
class Outer {
  private static void outerMethod() {
    System.out.println("inside outerMethod");
  }
      // A static inner class
  static class Inner {
    public static void main(String[] args) {
      System.out.println("inside inner class Method");
      outerMethod();
    }
  }
}
```

Output
inside inner class Method
inside outerMethod

Inner class

- 4) **Anonymous inner classes**
  Anonymous inner classes are declared without any name at all.

```
class Demo {
  void show() {
    System.out.println("i am in show method of super class");
  }
}
class Flavor1Demo {

  //  An anonymous class with Demo as base class
  static Demo d = new Demo() {
    void show() {
      super.show();
      System.out.println("i am in Flavor1Demo class");
    }
  };
  public static void main(String[] args){
    d.show();
  }
}
```

Output
i am in show method of super class
i am in Flavor1Demo class

In the above code, we have two class Demo and Flavor1Demo. Here demo act as super class and anonymous class acts as a subclass, both classes have a method show(). In anonymous class show() method is overridden.