UNIT-II

CLASSES, INHERITANCE, POLYMORPHISM

Classes and Objects: Classes, Objects, Methods, Constructors- Types of Constructors, Cleaning up unused objects- Garbage Collector, static keyword, this keyword, arrays, Command line arguments, Nested classes

Strings: String, StringBuffer, StringTokenizer

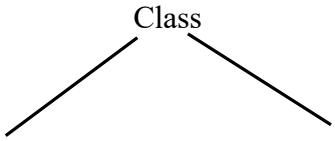
Inheritance and Polymorphism: Types of inheritance, deriving classes using extends keyword, super keyword, polymorphism – method overloading, constructor overloading, method overriding, final keyword, abstract classes

Classes and Objects:

- Class is a template or blueprint of an Object
- Object is an instance of theClass.

A Class in Java contains only 2 parts

1. State Part
2. Behaviour Part



1. State 2. Behaviour

Variables / Properties Functions / Methods

A Class in Java may contain

- Only State part
- Only Behaviour part
- Both State and Behaviour parts

Syntax of class:

```
class class_name
 {
 State – Variables / Properties
                    Behaviour – Methods / Constructors
}
```

State part and Behaviour part together are called members of the Class

Classes and Objects:

- Members of the Class are accessed using the Objects
- First Class is created,then Objects are created
- Class is created or defined only once.
- For the Class, any number of Objects can be created

Example for Class

```
class student
{
int rno;
String name;   State - Variables/Properties
float marks;

void display(){     Behaviour - Methods/Constructors
      ------------
}
}
```

Syntax for Object:

```
Class_namereference_varibale = new class_name();
```

Example for Object:

```
Student s1= new Student();
s1.rno=1;
                    s1.marks=100;
                    s1.display();
Student s2 =new Student ();
                    s2.rno=2;
                    s2. marks=98;
                    s2.display();
```

Note:

```
Save the file:  class_name.java
Compile: javacclass_name.java  (javac is java compiler )
Execute:    java class_name  ( java is Java Virtual machine – JVM )
```

Methods:

- A function which is written inside the class is called a method
- Java methods are blocks of code that perform a specific task
- A method allows us to reuse code, improving both efficiency and organization.
- All methods in Java must belong to a class.

Syntax for Method/Function:

```
Method / Function Definition
returnTypemethodName ([Parameters]) // Parameters - optional
{
     method body
  [return value]    // return value -  optional
}
```

Examples for Methods /Functions:

```
// Function/Method Definition
void display()
{
System.out.println( "RollNo is: " +rollno);
System.out.println( "Marks are:"+ marks);
}
// Function Call
display();
```

```
// Function/Method Definition
int sum( int a, int b)
{
     int c = a+b;
return  c;
 }
// Function Call
sum( 10,20));
```

```
// Function/Method Definition
voidsum(double a, double int b)
{
     int c = a+b;
System.out.println( " sum is " +c );
 }
// Function call
Sum(9.6 , 5.5);
```

1. Java program with one class:

```
class Student extends Object
        { int rno;
        float marks;
        void display() {
                System.out.println("rno is " + rno);
                System.out.println("marks are " + marks);
        }
        public static void main(String[] args)
        { Student s1 = new Student();
        s1.rno = 1;
s1.marks = 100;
                s1.display();
                Student s2 = new Student();
        s2.rno = 2;
                s2.marks = 98;
        s2.display();
        }
}
```

Output:

```
C:\nrcm>javac Student.java

C:\nrcm>java Student
rno is 1
marks are 100.0
rno is 2
marks are 98.0
```

2. Java program with two classes:

```
class Student extends Object
        { int rno;
        float marks;
         void display() {
                System.out.println("rno is " + rno);
                System.out.println("marks are " + marks);
        }
}
class StudentDemo extends Object {
         public static void main(String[] args)
                  { Student s1 = new Student();
                  s1.rno = 1;
                  s1.marks = 100;
                  s1.display();
                  Student s2 = new Student();
                  s2.rno = 2;
                  s2.marks = 98;
                  s2.display();
```

```
    }}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
rno is 1
marks are 100.0
rno is 2
marks are 98.0
```

Constructors:

Rule1: Name of the Constructor is same as thename of the class

Rule2: Constructors do not have return type

Rule 3: The first statement inside each Constructor must be either super() or this().

Bydefault, it is super ()

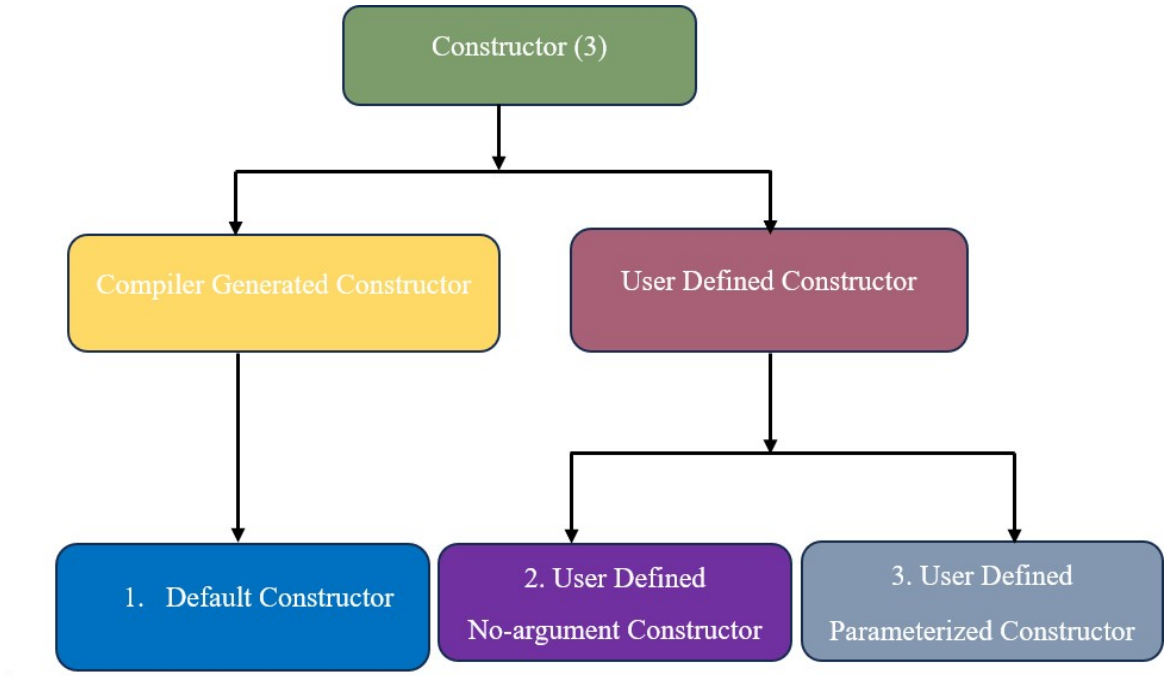
Rule 4: Constructors can have access modifiers

Rule 5: Constructors cannot be static, abstract or final

Purpose of the Constructor: The main purpose of the constructor is to initialize the instance variables of the object.

Types of Constructors: There are **3 types** of Constructors in Java. They are

1. **Default Constructor**
2. **User Defined No-arg Constructor**
3. **User Defined Parameterized Constructor**



Initialization of an object:

- Inside each object, instance variables of the class will be stored.
- Initialization of object is nothing but initialization of instance variables.
- Objects are stored in heap memory, therefore instance variables are also stored in heap memory.

Different ways to initialize the objects:

1. Default Constructor
2. User Defined No-arg Constructor
3. User Defined Parameterized Constructor
4. Direct Initialization
5. Copying the Reference
6. Using Reference Variables
7. User Defined No-argMethod
8. User Defined Parameterized Method

1. Default Constructor:

If there is no constructor in the class, then the compiler will provide **default constructor** for the class during the compilation time.
If there is atleast one constructor in the class, then the compiler will not provide default constructor for the class.

- Default constructor is no-arg constructor.Default constructor will assign default values for the instance variables based on the datatype.
- It is **called automatically** when an object is created.

Syntax:

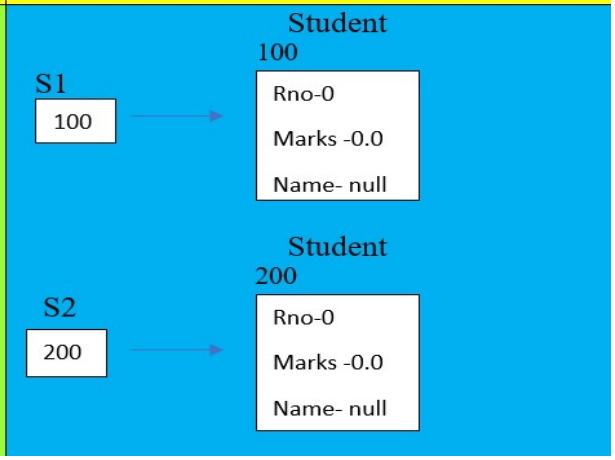
```
Student (){
      //default values are assigned to instance variables based on the datatype
 }
```

Program 1: Java Program on Default Constructor

```
// Default Constructor
class Student {
            int rno; // rno is instance variable
            String name;  // name is instance variable
            float marks;  // marks is instance variable
            // Default Constructor-Compiler Generated Constructor
            // Compiler will add this during compilation time
           Student() {
                    rno = 0;
                      name = null;
                      marks = 0.0f;
             }
             void display() {
                    System.out.println("RNO IS: " + rno);
                    System.out.println("NAME IS: " + name);
                    System.out.println("MARKS ARE: " + marks);
             }
}
class StudentDemo {
             public static void main(String[] args)
                     { Student s1 = new Student();
                     Student s2 = new Student();
                     s1.display();
                     s2.display();
             }
}
```

Output:

```
C:\nrcm>javac DefaultConstructor.java

C:\nrcm>java DefaultConstructor
RNO IS:0
NAME IS:null
MARKS ARE:0.0
RNO IS:0
NAME IS:null
MARKS ARE:0.0
```

Program	Initialization of object/ instance variables
Default constructor	

2. **User Defined No-argConstructor:**

Userdefined no-argument constructor is a constructor written by the programmer that does not take any parameters.

It is used to initialize the objects with the values chosenby the programmer.

It is **called automatically** when an object is created.

It allows different objects to be initialized with same values.

Program 2: Java Program on User Defined No-arg Constructor

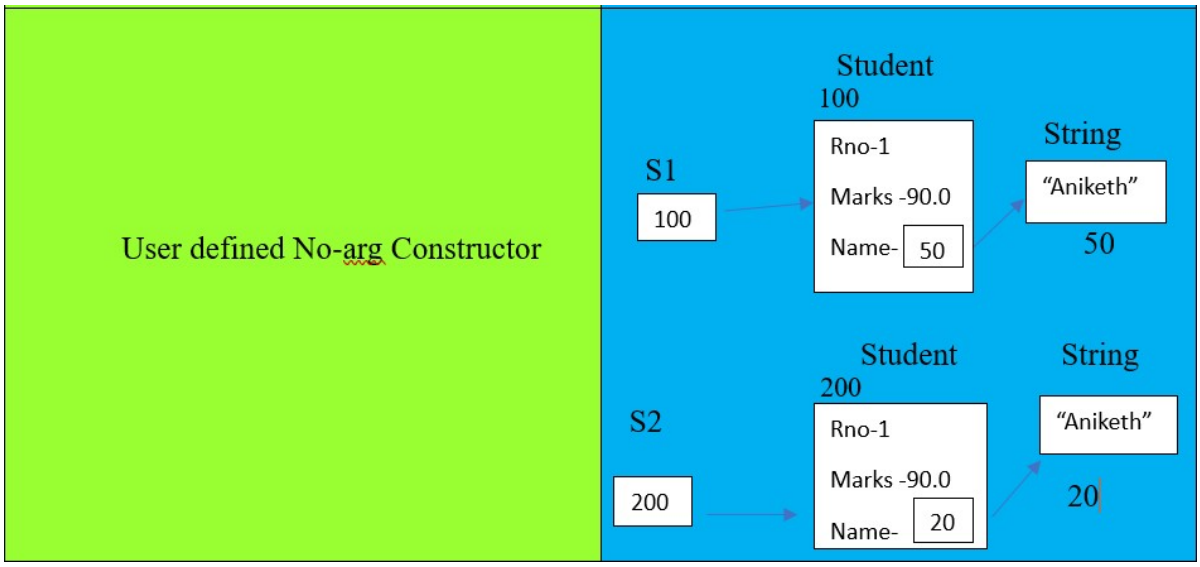
```
//User Defined No-argConstructor
class Student {
        int rno; // rno is instance variable
        String name;// name is instance variable
        float marks; // marks is instance variable
        // user defined no-arg constructor
      Student() {
            rno = 1;
              name = "Aniketh";
              marks = 90.0f;
      }
      void display() {
            System.out.println("RNO IS: " + rno);
            System.out.println("NAME IS: " + name);
            System.out.println("MARKS ARE: " + marks);
      }
}
class StudentDemo {
      public static void main(String[] args)
            { Student s1 = new Student();
            Student s2 = new Student();
```

```
        s1.display();
                s2.display();
         }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
RNO IS:1
NAME IS:Aniketh
MARKS ARE:90.0
RNO IS:1
NAME IS:Aniketh
MARKS ARE:90.0
```

Note: Drawback of user defined No-arg constructor - All the objects are given same values for instance variables.



3. User Defined Parameterized Constructor:

A parameterized constructor is a constructor that accepts one or more parameters. It is defined by the programmer (user-defined) to initialize objects with specific values. It is called automatically when an object is created. It allows different objects to be initialized with different values.

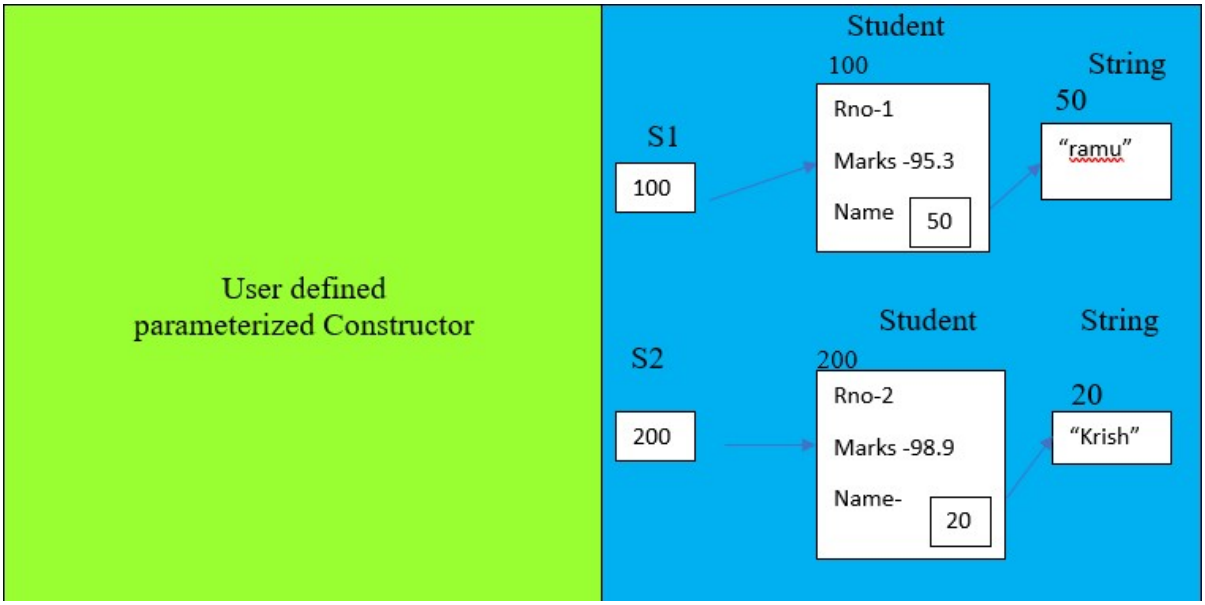
Program 3: Java Program on User Defined Parameterized Constructor

```
// User Defined Parameterized Constructor
class Student {
        int rno;
        String name;
        float marks;
```

```
        // User defined parameterized constructor
        Student(int rno1, String name1, float marks1) {
                rno = rno1;
            name = name1;
                marks = marks1;
         }
 void display() {
                System.out.println("RNO IS: " + rno);
                System.out.println("NAME IS: " + name);
                System.out.println("MARKS ARE: " + marks);
        }
}

class StudentDemo {
        public static void main(String[] args) {
                Student s1 = new Student(1, "ramu", 95.3f);
                Student s2 = new Student(2, "krish", 98.9f);
                s1.display();
                s2.display();
        }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
RNO IS:1
NAME IS:ramu
MARKS ARE:95.3
RNO IS:2
NAME IS:krish
MARKS ARE:98.9
```



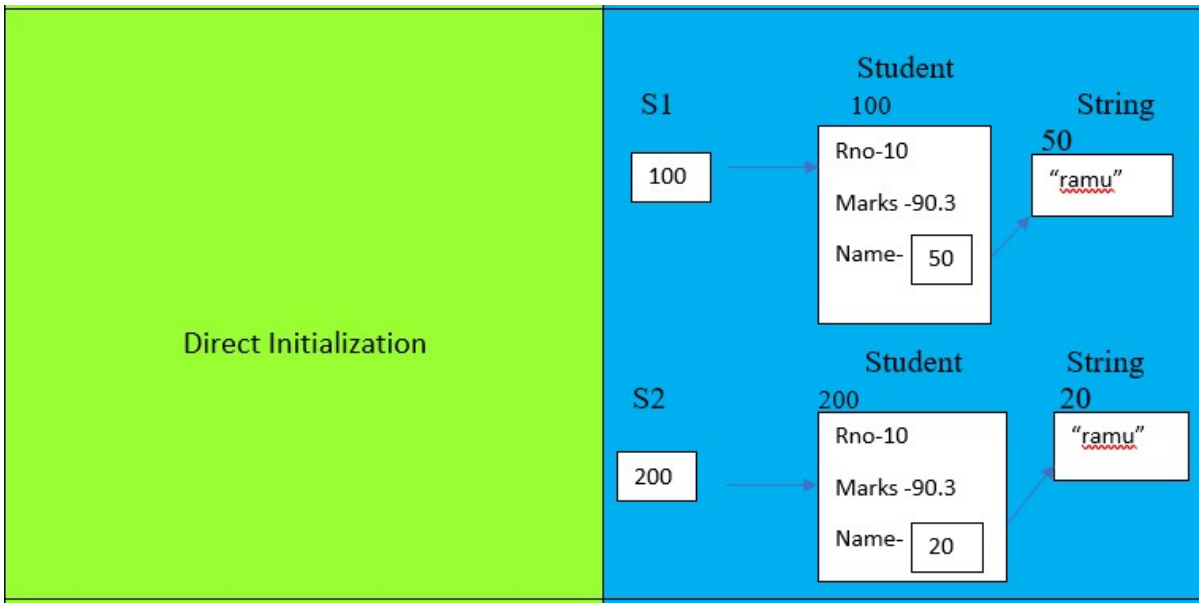
4. **Direct Initialization:**
   At the time of declaring the instance variables, we can initialize them. In case of default constructor, the compiler uses these initialized values in place of default values.

Program 4: Java Program using Direct Initialization

```
// Direct Initialization
class Student {
        int rno = 10;
        String name = "ramu";
        float marks = 90.3f;
                void display() {
                        System.out.println("RNO IS: " + rno);
                        System.out.println("NAME IS: " + name);
                        System.out.println("MARKS ARE: " + marks);
                }
}

class StudentDemo {
        public static void main(String[] args)
                { Student s1 = new Student();
                Student s2 = new Student();
                s1.display();
                 s2.display();
        }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
RNO IS:10
NAME IS:ramu
MARKS ARE:90.3
RNO IS:10
NAME IS:ramu
MARKS ARE:90.3
```



5. Using Reference variables:

In Java, reference variables are used to refer to object. A reference variable stores the address of an object, not the object itself.

A reference variable can point to only one object at a time, but multiple reference variables can point to the same object.Changing the reference variable does not change the object.

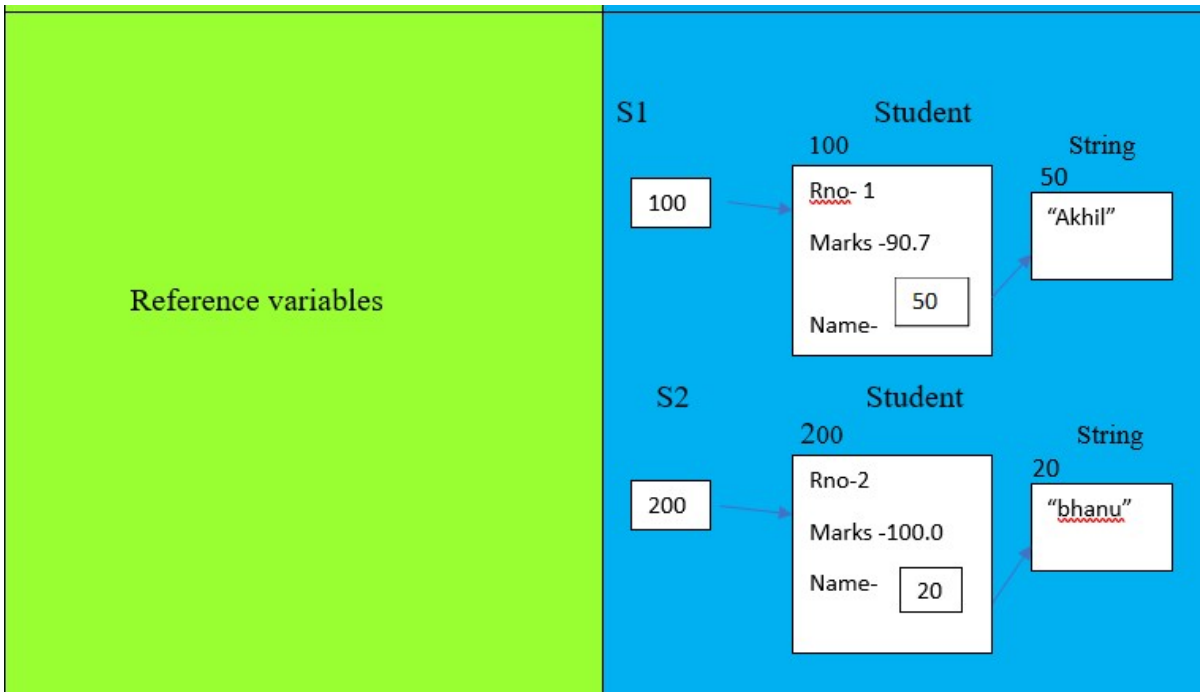
Program 5: Java ProgramUsing Reference variables:

```
//Reference variables
class Student {
   int rno;
   float marks;
   String name;
}
class StudentDemo{
public static void main(String[] args)
 { Student s1=new Student();
```

```
    Student s2=new Student();
    s1.rno=1;
    s1.marks=90.7f;
    s1.name="Akhil";
    s2.rno=2;
    s2.marks=100;
    s2.name="bhanu";
    System.out.println(s1.rno+" "+s1.marks+" "+s1.name);
    System.out.println(s2.rno+" "+s2.marks+" "+s2.name) ;|
    }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
1 90.7 Akhil
2 100.0 bhanu
```



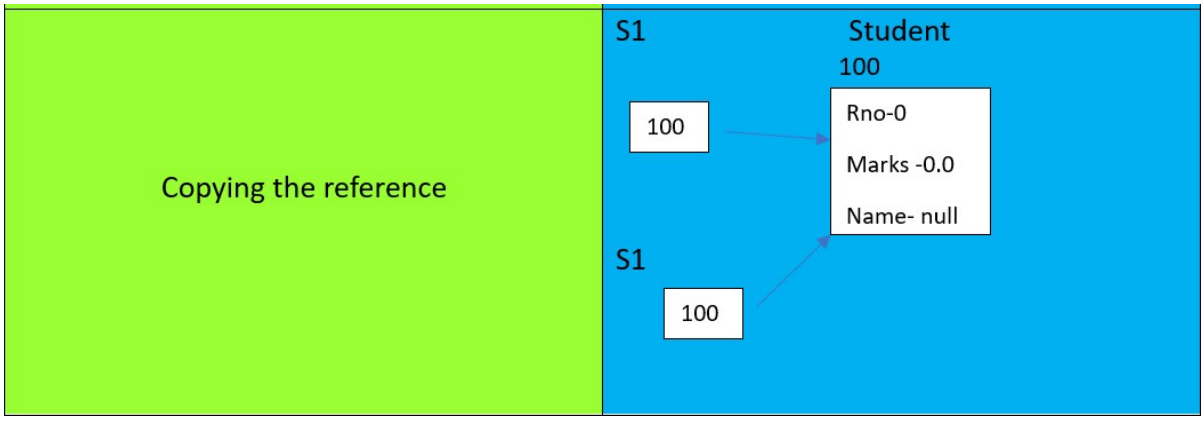
6. **Copying the reference**:
   Copying a reference variable means assigning one reference variable to another.
   In this case, no new object is created both reference variables point to the same object.
   Changes made using one reference variable are reflected through the other reference variable. Memory is used efficiently because only one object exists in the heap.
   If one reference variable is assigned null, the other reference variable still points to the object.

Program 6: Java Program onCopying the reference variable

```
// Copying the Reference
class Student {
        int rno;
        String name;
        float marks;
        void display() {
                System.out.println("RNO IS: " + rno);
                System.out.println("NAME IS: " + name);
                System.out.println("MARKS ARE: " + marks);
        }
}

class StudentDemo {
        public static void main(String[] args)
                { Student s1 = new Student();
                Student s2 = s1;  // s2 references the same object as s1
                s1.display();
                s2.display();
        }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
RNO IS:0
NAME IS:null
MARKS ARE:0.0
RNO IS:0
NAME IS:null
MARKS ARE:0.0
```



7. User Defined No-argMethod:

A **user-defined no-argument method** is a method created by the programmer that does not accept any parameters. It is used to initialize an object's instance variables with values provided by the user or programmer after the object is created. Unlike a constructor, it can be called **multiple times** whenever reinitialization is needed.

The object must first be created using the `new` keywordand then the initialization method is called. It provides flexibility because the programmer can decide when to initialize or reset the object's data. It is useful when initialization should happen separately from object creation.Different objects can be initialized with same values.

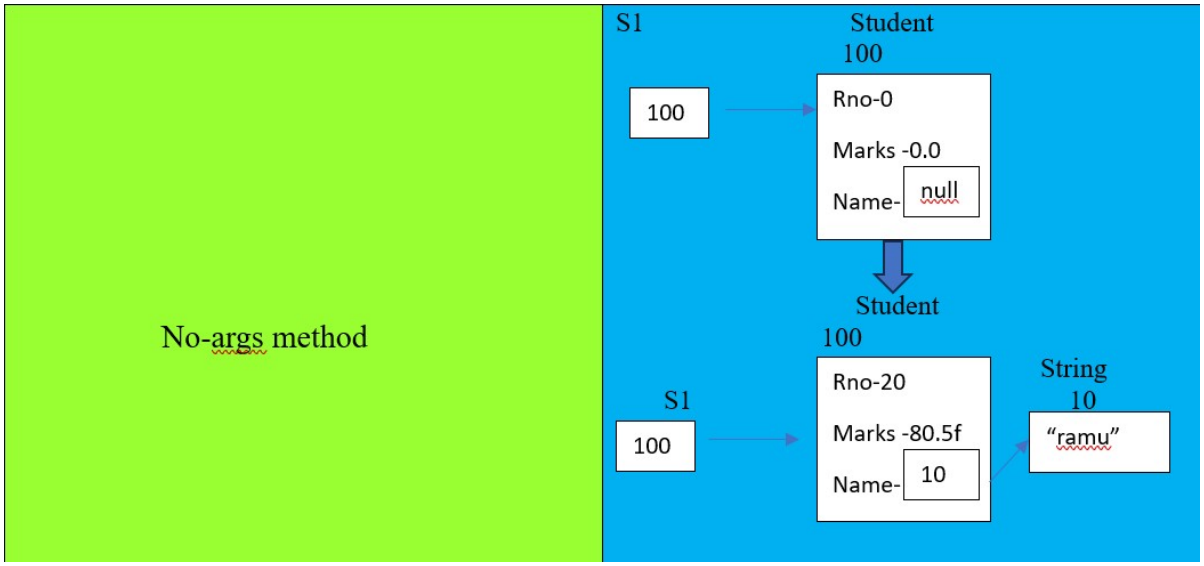
Program 7: Java Program using User Defined No-arg Method

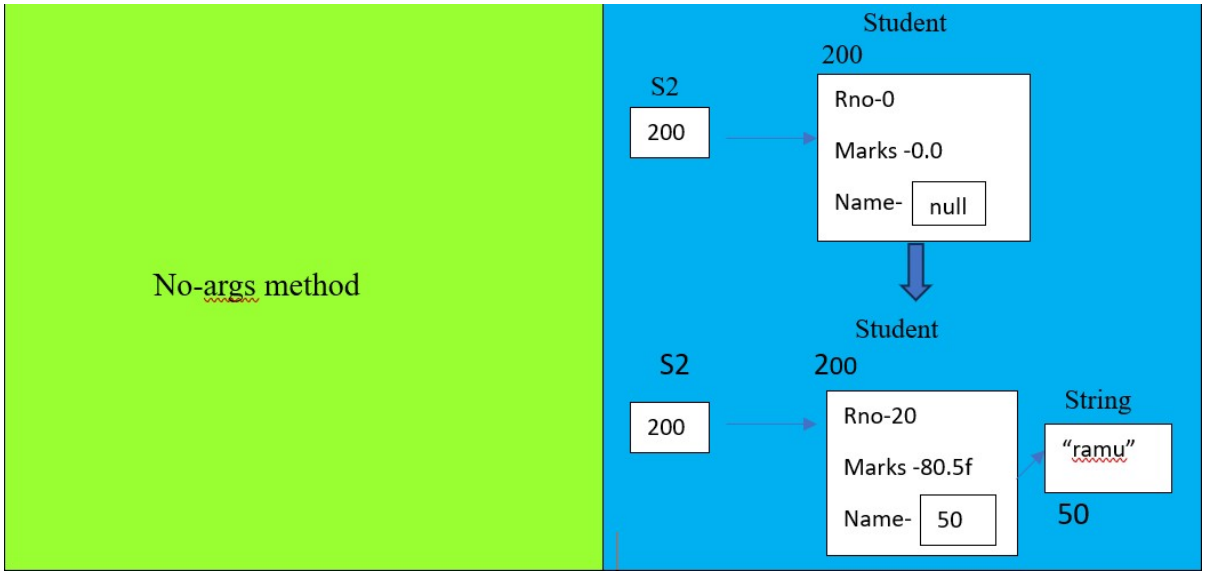
```
// User Defined No-argMethod
class Student {
        int rno;
        String name;
        float marks;
        void setDetails() {
                rno = 20;
                marks = 80.5f;
                name = "ramu";
        }
        void displayDetails() {
                System.out.println("RNO IS: " + rno);
                System.out.println("NAME IS: " + name);
                System.out.println("MARKS ARE: " + marks);
        }
}

class StudentDemo {
        public static void main(String[] args)
                { Student s1 = new Student();
                Student s2 = new Student();
                s1.setDetails();
                s2.setDetails();
                s1.displayDetails();
                s2.displayDetails();
        }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
RNO IS:20
NAME IS:ramu
MARKS ARE:80.5
RNO IS:20
NAME IS:ramu
MARKS ARE:80.5
```

Note:in this case,first the default constructor is executed and then the method is executed.





8. User Defined Parameterized Method:

A **user-defined parameterized method** is a method created by the programmer that accepts one or more parameters. It is used to initialize an object's instance variables with values provided by the user or programmerafter the object is created.

Different objects can be initialized with **different values** by passing different arguments to the method. The object must first be created using the **new** keyword and then the parameterized method is called.

It provides greater flexibility than a no-argument method because values are supplied at runtime. The method can be called multiple times to update or reinitialize an object's data.

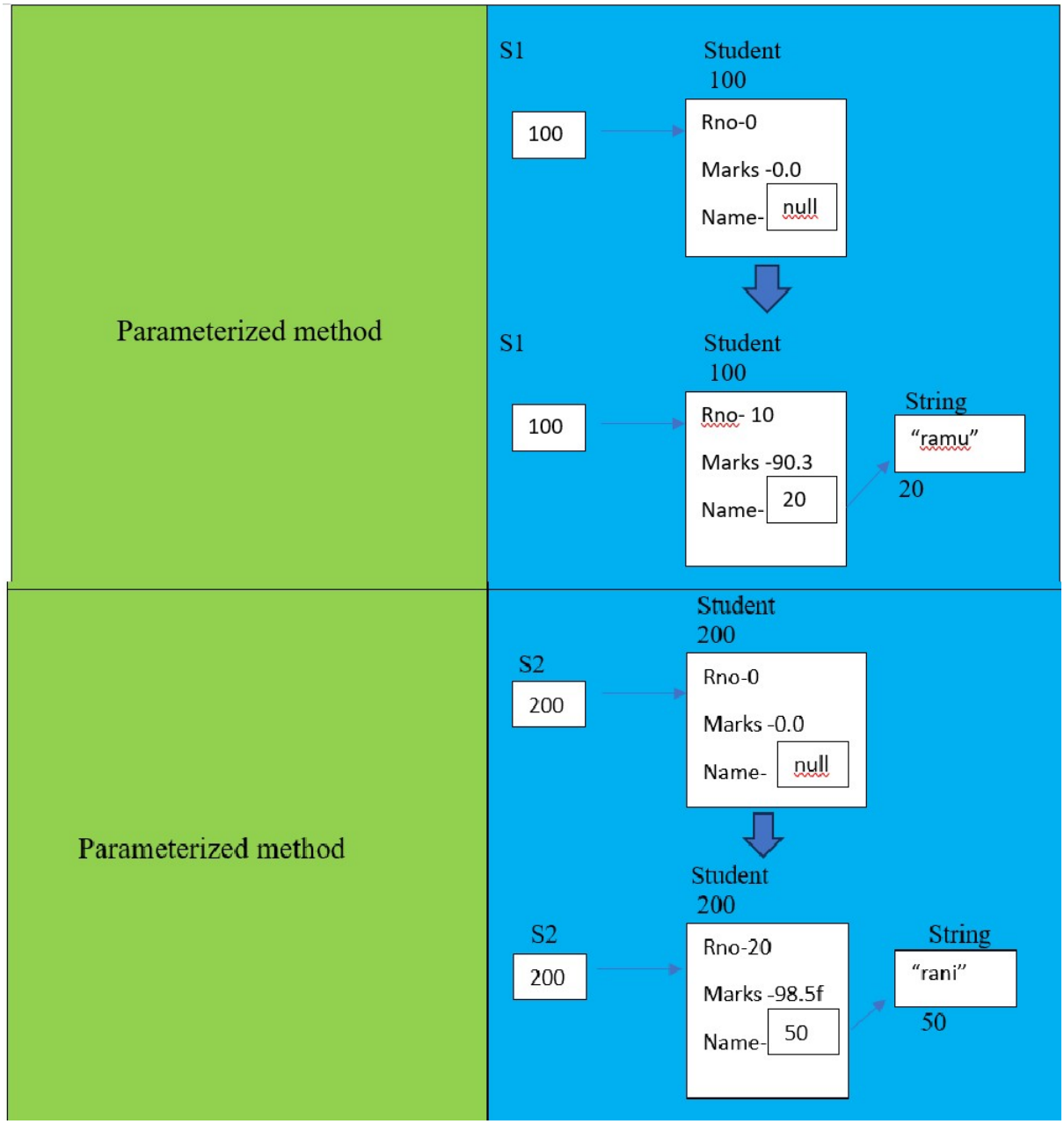
Program 8: Java Program using User Defined Parameterized Method

```
//Paramterized method
class Student{
int rno;
float marks;
String name;
void setDetails(int rno, float marks,String name)
{ this.rno=rno;
this.marks=marks;
this.name=name;
        }
void displayDetails()
{ System.out.println("RNO IS:"+rno);
System.out.println("NAME IS:"+name);
System.out.println("MARKS ARE:"+marks);
        }
}
class StudentDemo{
public static void main(String[] args)
{ Student s1=new Student();
Student s2=new Student();
s1.setDetails(10,90.3f,"ramu");
s2.setDetails(20,98.5f,"rani");
s1.displayDetails();
s2.displayDetails();
        }
}
```

Output:

```
C:\nrcm>javac StudentDemo.java

C:\nrcm>java StudentDemo
RNO IS:10
NAME IS:ramu
MARKS ARE:90.3
RNO IS:20
NAME IS:rani
MARKS ARE:98.5
```

Note: in this case,first the default constructor is executed and then the method is executed.



Garbage Collection

- In languages like C++ and C, the programmer is responsible for both the creation and destruction of objects.

- Usually, the programmer takes great care while creating objects but neglects their destruction. Due to this negligence, at a certain point, memory may not be available for the creation of new objects and there may be a maximum chance of failing applications due to memory problem.
- In Java, the programmer is only responsible for the creation of objects. Java developers provided an assistant called the **Garbage Collector (GC)**, which runs continuously in the background.
- Because of the Garbage Collector, there is less chance of a Java application failing, which is why Java is considered a **Robust language**.

Types of Objects

Objects in memory are categorized into two types:

1. **Useful Objects:** When there is an active reference variable pointing to an object, it is considered a useful object.
    - *Example:* Student s1 = new Student();
2. **Useless Objects:** When there is no reference variable pointing to an object, it is considered a useless object.
    - *Example:*new Student();

      An object whose reference has been set to null or has gone out of scope.

Key Rule: The Garbage Collector destroys **only** useless objects.

- When JVM invokes the Garbage Collector, it enters the heap memory to destroy useless objects. But we don't know exactly how many objects are destroyed when it enters in to the heap memory. The programmer cannot directly invoke the Garbage Collector, but the programmer can r**equest** the JVM to run Garbage Collector

Programmer ---> JVM ---> Garbage Collector

Objects eligible for Garbage Collection

```
class Student {
        int no;
        float marks;
        public static void main(String[] args)
                { Student s1 = new Student();
                Student s2 = new Student();
                s1 = null;    // 1 object is now eligible for GC
                s2 = null;    // 2 objects are now eligible for GC

        }
}
```

Methods to request JVM to run Garbage Collector

The programmer cannot invoke Garbage Collector directly, but the programmer can send a request programmatically to JVM to run Garbage Collector using following two approaches:

1. Using System class – gc() method

The predefined class System contains a static method called gc().

Using System.gc(), we are sending a request to JVM to run Garbage Collectorbut we don't know whether our request is accepted by the JVM or notand we also don't know whether Garbage Collectordestroys all the useless objects.

Program1: Java Program on Garbage Collection using System.gc() method

```
class Student {
        int no;
        float marks;
        public static void main(String[] args) {
                // Creating objects
                Student s1 = new Student();
                Student s2 = new Student();
                // Making objects eligible for garbage collection
                s1 = null;
                s2 = null;
System.out.println("Requesting Garbage Collection...");
System.gc(); // Sending a request to JVM to run Garbage Collector
        System.out.println("Program execution completed.");
          }
  }
```

Output:

```
C:\nrcm>javac Student.java

C:\nrcm>java Student
Requesting Garbage Collection...
Program execution completed.
```

2. Using the Runtime class – gc() method

Runtime is a predefined class and it is a **Singleton class**.

- For singleton class, we cannot create objects using new operator
  Ex: Runtime r = new Runtime(); // Compiler Error
- To createan object for Runtime class, we must use a static factory method called getRuntime().

Runtime Class - Methods

```
class Runtime {
```

```
        static Runtime getRuntime()
{ returnobj;
        }
        void gc() { }                   // Non-static method
        long totalMemory() { }   // Non-static method
long freeMemory() { }    // Non-static method
}
```

- totalMemory(): Used to find the total size of the heap.
- freeMemory(): Used to find thefree memory available in the heap.

Program2: Java Program on Garbage Collection using Runtime class – gc() method

```
import java.util.Date;
class RuntimeDemo {
 public static void main(String[] args)
  { Runtime r = Runtime.getRuntime();
System.out.println("Total Memory = " + r.totalMemory());
System.out.println("Free Memory Before Object Creation = " + r.freeMemory());

  for (int i = 0; i< 10000; i++)
{ Date d = new Date();
d = null;    // Object becomes eligible for garbage collection
}
System.out.println("Free Memory After Object Creation = " + r.freeMemory());
r.gc();    // Request JVM to perform Garbage Collection
System.out.println("Free Memory After Garbage Collection = " + r.freeMemory());
}
}
```

Output:

```
C:\nrcm>javac RuntimeDemo.java

C:\nrcm>java RuntimeDemo
Total Memory = 532676608
Free Memory Before Object Creation = 530159648
Free Memory After Object Creation = 529656312
Free Memory After Garbage Collection = 16140584
```

Finalization andfinalize() Method

Just before destroying any object, the Garbage Collector invokes the finalize() method on that object to perform **cleanup activities** such as closing database connections/network connections/file streams

If we want to perform cleanup activities like closing connections, we must override the finalize() method of the Object class. Otherwise, there is no need to override finalize() method of the Object class

Just before destroying the object, the garbage collector executes either the default null implementation of thefinalize() method of the Object class or our overriding version of finalize() method

It is automatically executed by Garbage Collector, there is **no need to invoke it explicitly**

finalize() method in Object class

```
class Object {
protected void finalize() throws Throwable {
// Null/empty implementation
}
}
```

Program 3: Java Program to overridefinalize() method of Object class

```
import java.io.File;
class Connection {
 private static Connection conn = new Connection();
public static Connection getConnection() {
return conn;
}
public void close()
{ System.out.println("Connection
closed.");
}
}
class Test1 {
Connection c = Connection.getConnection();
File f = new File("abc.txt");
      // Overriding finalize() method
protected void finalize() throws Throwable {
System.out.println("finalize() method executed");
c.close();
//super.finalize();
}
```

```
Test1 t1 = new Test1();
t1 = null;    // Object becomes eligible for garbage collection

System.gc();  // Request JVM to run GC
System.out.println("Main method finished");
}
}
```

Output:

```
c:\nrcm>javac Test1.java
Test1.java:15: warning: [removal] finalize() in Object has been deprecated and marked for removal
          protected void finalize() throws Throwable {
                         ^
1 warning

c:\nrcm>java Test1
Main method finished
finalize() method executed
Connection closed.
```

Command line Arguments:

Command line arguments are values passed to a Java program when it is executed.
They allow users to provide input without using the keyboard during program execution

Syntax: Command line arguments are received through the main() method

public static void main(String[] args):

- **args** is an array of String objects, arguments are Strings
- Every command line argument is stored as a String, even if it represents a number
- Strings are converted in to numeric values the methods like:

int n = Integer.parseInt(args[0]);

- Arguments are accessed using array indexes

System.out.println(args[0]); // First argument
System.out.println(args[1]); // Second argument

- The number of arguments is obtained using **args.length**

Programwithout using command line arguments

```
//without command line arguments.
class Sample{
public static void main(String [] args)
 { int a=10, b=20;
System.out.println("sum of a and b is" + (a+b));
}
}
```

Output:

```
C:\nrcm>javac Sample.java

C:\nrcm>java Sample
sum of a & b is30
```

Program using command line arguments:

```
//with command line arguments
class Sample1{
public static void main(String [] args)
{ int a= Integer.parseInt(args[0]);
int b= Integer.parseInt(args[1]);
System.out.println("Sum of a & b is" + (a+b));
 }
}
```

Output:

```
C:\nrcm>javac Sample1.java

C:\nrcm>java Sample1 10 20
Sum of a & b is30
```

Note: Integer is the wrapper class corresponding to primitive datatype int.

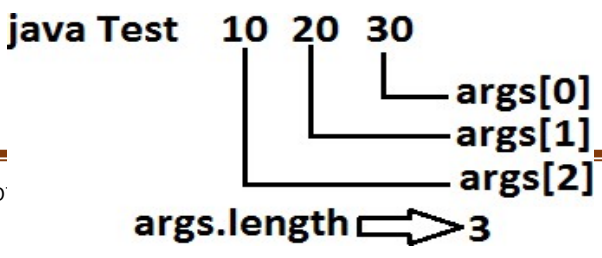
```
class Integer {
static int parseInt (" “) {
                        return int;
}
}
```

Except for char and boolean, all the other 6 numerical datatypes (byte, short, int, long, float, double) have wrapper classes.

```
Integer.parseInt (args [0]);
```

Since parseInt( ) is **a static** method of Integer wrapper class and we are accessing it in the Sample class i.e., outside the class, so we must use **classname. method name.**

Ex: Integer.parseInt( ), Float.parseFloat( ), Byte.parseByte( )etc.



- Numerical Wrapper classes: Byte, Short, Integer, Long, Float, Double
- Parse methods are used to convert strings into corresponding datatypes

Static keyword:

There are four ways to use the static keyword:

1. Static blocks
2. Static variables
3. Static methods
4. Static import (packages)

1. Static blocks

blocks in Java

**/ **

init static

Syntax for static block:

```
static
{
initializes thestatic variables
}
```

Syntax for init block:

```
{
   initializes the instance variables
}
```

2.Static Variables: Static variables are the variables that belong to the class rather than to individual objects. They can help reduce memory usage when used appropriately. Only once memory is allocated for static variables and static blocks i.e. before the main() method execution and can be accessed any number of times.

Key points about static variables:

- **One copy per class**: Only a single copy of a static variable exists, regardless of how many objects are created.
- **Shared by all objects**: Every object of the class accesses the same static variable. Memory for a static variable is allocated only once when the class is loaded into memory.
- **Access without creating objects**: Static variables can be accessed with class name.

Static Variables vs. Instance Variables

Static Variables	Instance Variables
1. Static variables are defined/placed inside the class and outside themethod/block/constructor and declared with static keyword.	1. Instance variables are defined/placed inside the class and outside the method/block/constructorand not declared with static keyword.
2. Static variables are class variables, i.e., the scope of static variables is throughout the class.	2. Instance variables are object level variables, i.e., the scope of instance variables is within the object.
3. Memory is allocated for static variables during the class loading time and released during class unloading time	3. Memory is allocated for instance variables at the time of object creation and released at the time of object destruction
4. Memory for static variables is allocated only once for the entire class.	4. Memory for instance variables is allocated for each object.
5. Static variables are stored in method area memory	5. Instance variables are stored in heap memory because instance variables are part of objects and objects are stored in heap memory
6. Common properties are declared as static variables.	6. Different properties are declared as instance variables.

Programs on Static Variables:

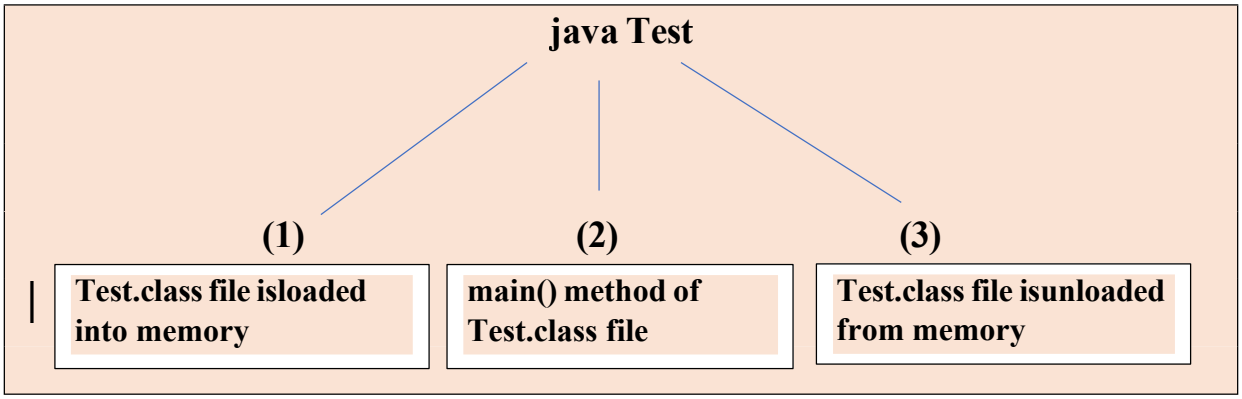
- How to use static variables within the same class (3 ways):
  - With class name
  - With object
  - Direct
- How to use static variables outside the class (2 ways):
  - With class name
  - With object

Compilation:

$ javacTest.java —>Test.class file is created.

Execution:

$ java Test (JVM —>loads / verifies / executes)



During class loading time all the static variables and static blocks will be executed in the order in which they appear i.e. from top to bottom.

Program 1: Java Program to access static variables within the same class

```
// within the same class.
class Student {
        static int collegecode = 20; // before main() method
        static String collegename = "NRCM"; // before main() method
        int rno;
        String name;
        public static void main(String[] args) {
                // With class name
System.out.print(Student.collegecode+" ");
System.out.println(Student.collegename);
                // Directly
System.out.println(collegecode + " " + collegename);
                Student s1 = new Student();
                // With object
System.out.println(s1.collegecode + " " + s1.collegename);
System.out.println(s1.rno + "  " + s1.name);
        }
}
```

Output:

```
C:\nrcm>javac Student.java

C:\nrcm>java Student
20 NRCM
20 NRCM
20 NRCM
0  null
```

Program 2:Java Program to access static variablesoutside the class

```
class Student
{ int rno;
String name;
        static String collegename = "NRCM";
static int collegecode = 20;
}
class Test {
public static void main(String[] args) {
// Accessing static variables outside the class via Class Name
System.out.println(Student.collegecode + " " + Student.collegename);
// Creating an object of Student class
Student s1 = new Student();
// Accessing static variables via Object Reference
System.out.println(s1.collegecode + " " + s1.collegename);
// Accessing instance variables via Object Reference
System.out.println(s1.rno + " " + s1.name);
}
}
```

Output:

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
20 NRCM
20 NRCM
0 null
```

Note:

1. Why should we declare common properties as static? If we declare common properties as static, the memory is allocated for the static variables only once, and all the objects of that class can use the static variable.
2. If we won't declare them as static, then memory will be allocated for each object since they are treated as instance variables. Hence, memory is wasted.

Rule: If we change a static variable with any one of the objects, then the change is reflected across all the objects.

Program 3: Java Program on static variables

```
class Student {
        int rno;
        String name;
        static String collegename = "NRCM";
        static int collegecode = 20;
}
class Test {
        public static void main(String[] args) {
                // 1. Accessing via Class Name directly
System.out.println(Student.collegecode + " " + Student.collegename);
                // 2. Creating first object (s1)
                Student s1 = new Student();
System.out.println(s1.collegecode + " " + s1.collegename);
                // 3. Creating second object (s2)
                Student s2 = new Student();
System.out.println(s2.collegecode + " " + s2.collegename);
                // 4. Modifying static variable 'collegecode' using s1 reference
                s1.collegecode = 40;
System.out.println(s2.collegecode); // Change is visible via s2
               // 5. Modifying static variable 'collegename' using s2 reference
               s2.collegename = "NRCMENG";
System.out.println(s1.collegename); // Change is visible via s1
        }
}
```

Output:

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
20 NRCM
20 NRCM
20 NRCM
40
NRCMENG
```

3. Static Methods:

A static method belongs to the class rather than any particular object. It' is used when the method doesn't depend on instance-specific data.

The main difference between static methods and instance methods is whether they belong to the class itself or to an object (instance) of the class.

Feature	Static Methods	Instance Methods
Belongs to	The class	An object (instance)
Requires an object?	No	Yes
Can access instance variables?	No (unless given an object)	Yes
Can access static variables?	Yes	Yes
Called using	ClassName.method()	object.method()

Program 4:Java Program on static methods

```
class Student {
        String name;      // instance variable
        static String college = "NRCM";     // static variable
        //  Constructor
        Student(String name) {
                this.name = name;
        }
        //  instance method
        void displayStudent() {
System.out.println("Name: " + name);
System.out.println("College: " + college);
        }
        //  static method
        static void displayCollege()
{ System.out.println("College: " + college);
        }
}
class Test {
        public static void main(String[] args) {
                // Calling static method  (no object needed)
Student.displayCollege();
                // Creating objects
                Student s1 = new Student("Rahul");
                Student s2 = new Student("Priya");
                // Calling instance methods
                s1.displayStudent();
                s2.displayStudent();
        }
}
```

Output:

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
College: NRCM
Name: Rahul
College: NRCM
Name: Priya
College: NRCM
```

thiskeyword:

this is a java keyword, it can be used to represent current class object.

In java applications, we are able to utilize **this** keyword in the following 4 ways

1. To refer current class variables
2. To refer current class methods
3. To refer current class constructors
4. To return current class object

1. To refer current class variables

- If we want to refer current class variable by using this keyword, then we have to use the following syntax

this.var_name

- In java applications, when we have same set of variables at local level and class level, to refer class level variables from the methods then we have to use “this” keyword.

```
class A{
     int i=100;
     int j=200;
A(int i , int
j){ System.out.println( i + " "+
j ); System.out.println( i + " " +
j );
    }
}
class ThisDemo{
      public static void main(String
            args[]){ A a=new A(10,20);
      }
```

Output:

```
C:\nrcm>javac ThisDemo.java

C:\nrcm>java ThisDemo
10 20
100  200
```

2. To refer current class methods

- If we want to refer current class method by using 'this' keyword then we have to use the following syntax.

this.method_name([param_list]);

Note: In java applications, to access current class methods, no need to create object and reference variables for current class, directly we can access current class method using this keyword is one option to access class methods.

```
class
A{ void
m1() {
System.out.println("m1-A");
 m2();
this.m2();
}
void m2()
{ System.out.println("m2-
A");
}
}
class ThisDemo2{
public static void main(String
args[]){ A a=new A();
 a.m1();
```

Output:

```
C:\nrcm>javac ThisDemo2.java

C:\nrcm>java ThisDemo2
m1-A
m2-A
m2-A
```

3. To refer current class constructors

- If we want to refer current class constructor by using "this"keyword,then we have to use the following syntax

this([parameter_list]);

- If we want to refer current class constructor by using "this" keyword then the respective "this" statement must be first statement otherwise compiler will raise an error
- If we want to refer current class constructor by using "this" keyword then the respective "this" statement must be provided in another current class constructor only, not from normal java methods (compiler will raise an error like call to this statement must be first statement in constructor)

```
class
A{ A()
{
this(10);
System.out.println("A-constructor");
}
A(int i)
{ this(2233.3f);
System.out.println("A-int-param-constructor");
 }
A(float f) {
System.out.println("A-float-param-constructor");
 }
}
class ThisDemo4{
public static void main(String args[])
{ A a=new A();
}
```

Output:

```
C:\nrcm>javac ThisDemo4.java

C:\nrcm>java ThisDemo4
A-float-param-constructor
A-int-param-constructor
A-constructor
```

4. To return current class object

- If we want to return current class object by using "this" keyword then we have to use the following syntax.

return this;

- If we access all the current class methods by returning "this" keyword then it is called as method chaining.

```
class
A{ Am
1() {
System.out.println("m1-A");
return this;
}
A m2() {
System.out.println("m2-A");
return this;
}
A m3() {
System.out.println("m3-A");
return this;
}
}
class ThisDemo5{
  public static void main(String args[])
{ A a= new A();
a.m1().m2().m3();
}
```

Output:

```
C:\nrcm>javac ThisDemo5.java

C:\nrcm>java ThisDemo5
m1-A
m2-A
m3-A
```

ARRAYS:

Arrays are objects in Java.

Every array has a public final instance variable named length i.e. length is an instance variable of an array object.

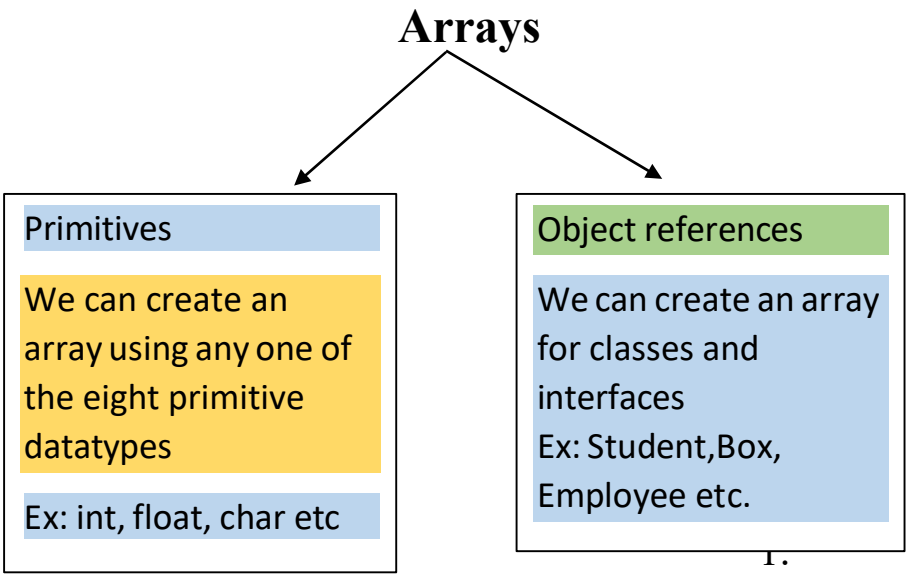
Array	Arrays class
A special Java language feature	A utility class in java.util package
Has the length instance variable	Has static methods like sort(), tostring() etc.

Example: arr.length	Example: Arrays.sort(arr)

Types of Arrays:

Based on datatype, there are two types of arrays in Java

1. Arrays of primitives
2. Arrays of object references



1. Arrays of primitives:

1. One Dimensional arrays:

int[] a; or int a[]; byte[] b; boolean[] c;

For arrays of primitives, we use the following three steps:

1. Declare
2. Create
3. Initialize

Ex: **Declare &Create:** int[] a = new int[5]; // a is reference variable
// a is a variable which is of integer type array

(OR)

Declare: int[] a;

Create: a = new int[5]; // Size must be given in RHS

- Inside each array object, length instance variable is stored
- Default values are stored in the array based on the datatype

Initialization of the arrays - 5 ways:

1. Direct initialization
2. for loop
3. Command Line Arguments
4. BufferedReader class
5. Scanner class

Prog1: Direct Initialization

```
class ArrayTest{
        public static void main(String[] args)
                { int []a = new int[4];
                //initialize the array - directly
a[0]=2;
a[1]=3;
a[2]=4;
a[3]=7;
System.out.println("Array elements are");
System.out.println(a[0]+" "+ a[1] +" "+ a[2] +" "+ a[3]);
        }
}
```

Output:

```
C:\nrcm>javac ArrayTest.java

C:\nrcm>java ArrayTest
Array elements are
2 3 4 7
```

Prog 2: Using for loop

```
class ArrayTest {
        public static void main(String[] args)
{ float[] f1= new float[4];
                // initializing the array using for loop
                // reading the array
for(int i=0; i< f1.length; i++) {
                        f1[i]= i + 1;
                }
                // writing the array
for(int i=0; i< f1.length; i++)
{ System.out.println(f1[i]);
```

```
        }
    }
}
```

Output:

```
C:\nrcm>javac ArrayTest.java

C:\nrcm>java ArrayTest
1.0
2.0
3.0
4.0
```

Prog 3: Using Command Line Arguments

```
class ArrayTest {
        public static void main(String[] args)
{ int[]  a = new int[4];
for(int i=0; i<a.length; i++) {
                    a[i]= Integer.parseInt(args[i]);
                }
for(int i=0; i<a.length; i++)
{ System.out.println(a[i]);
                }
        }
}
```

Output:

```
C:\nrcm>javac ArrayTest.java

C:\nrcm>java ArrayTest 4 7 2 1
4
7
2
1
```

Prog 4: Using BufferedReader class (input through keyboard)

- BufferedReader class is a pre-defined class available in java.io package.

```
class BufferedReader
{
    String readLine(); // non-static method
```

```
}
```

- readLine() method is a non-static method, so it can be accessed using BufferedReader class object. It returns String which needs to be converted to the corresponding primitive datatype using parse methods.
- Creating BufferedReader class object

```
BufferedReaderbr = new BufferedReader(new InputStreamReader(System.in));

br.readLine(); // calling readLine() method
```

```
import java.io.*;
class ArrayTest {
        public static void main(String[] args) throws IOException
{ int[] a= new int[4];
BufferedReaderbr= new BufferedReader(new InputStreamReader(System.in));
for(int i=0; i<a.length; i++) {
                        a[i]= Integer.parseInt(br.readLine());
                }
System.out.println("Displaying elements");
for(int i=0; i<a.length; i++)
{ System.out.println(a[i]);
                }
        }
}
```

Output:

```
C:\nrcm>javac ArrayTest.java

C:\nrcm>java ArrayTest
4
7
2
1
Displaying elements
4
7
2
1
```

Prog 5: Using Scanner class

- Scanner is a pre-defined class. It is available in java.util package.
- **Methods in Scanner class:** (non-static)
    - byte nextByte()
    - short nextShort()

- int nextInt()
- long nextLong()
- float nextFloat()
- double nextDouble()
- String nextLine()

Scanner s = new Scanner (System.in);

```
import java.lang.*;
import java.util.*;
class ArrayTest {
        public static void main(String[] args)
{ int[] a = new int[5];
                Scanner sc = new Scanner(System.in);
                for (int i = 0; i<a.length; i++) {
System.out.println("Enter the value");
                        a[i] = sc.nextInt();
                }
System.out.println("Elements are");
                for (int i = 0; i<a.length;
i++){ System.out.println (a[i]);
                }
        }
}
```

Output:

```
C:\nrcm>javac ArrayTest.java

C:\nrcm>java ArrayTest
Enter the value
5
Enter the value
9
Enter the value
3
Enter the value
1
Enter the value
6
Elements are
5
9
3
1
6
```

Declare, create and initialize in one step

int[] a = {1,4,3}

Anonymous arrays:

No name for the array. We use anonymous arrays as method arguments.

Syntax: new int[]{1, 2, 4, 5}

```
// Program on Ananymous array
class ArrayTest{
	void m1 (int[] a) {
		for (int i = 0; i<a.length; i++)
{ System.out.println(a[i]);
		}
	}
	public static void main(String[] args)
{ ArrayTest t1 = new ArrayTest();
		t1.m1(new int[]{4, 5, 6, 7});
	}
}
```

Output:

```
C:\nrcm>javac ArrayTest.java

C:\nrcm>java ArrayTest
4
5
6
7
```

2. Two-Dimensional arrays:

1. Declare and Create 2-D array

int[][] a = new int [3] [2];

Two-Dimensional array is an array of one-dimensional arrays. In the above case, there are 3 one-dimensional arrays and size of each one-dimensional array is 2. Initially, default values are stored in the last level.

2. Initialization:
   - Direct initialization
   - for loops
   - Command line arguments
   - BufferedReader Class
   - Scanner Class

```
// Program on 2-D array
class TestArray{
      public static void main(String[] args)
{ int[][] a = new int [3][2];
                  Scanner sc = new Scanner(System.in);
System.out.println("Initializing the array ");
                  for (int i = 0; i<a.length; i++) {
                        for (int j = 0; j <a[0].length;
                              j++){ a[i][j] =
                              sc.nextInt();
                        }
                  }
System.out.println("Three 1-D arrays of size 2");
                  for (int i = 0; i<a.length; i++){
                        for (int j = 0; j <a[0].length;
j++){ System.out.print(a[i][j] + " ");
                        }
System.out.println();
                  }
      }
```

Output:

```
C:\nrcm>javac TestArray.java

C:\nrcm>java TestArray
Initializing the array
1 2
5 6
7 4
Three 1-D arrays of size 2
1 2
5 6
7 4
```

Jagged arrays:

To declare one-dimensional array of different sizes.

Syntax:

```
int[][] a = new int [3] [];
a[0] = new int[3];
a[1] = new int[5];
a[2] = new int[2];
```

3. Three-dimensional arrays:

```
int[][][] a = new int [3] [2] [2];
```

Three-dimensional array is an array of twodimensional arrays.
In the above case:
int [3] [2] [2]

- 3 - two dimensional arrays
- 2 - one dimensional arrays
- size of each one dimensional array is 2

```
import java.util.*;
class TestArray{
        public static void main(String[] args)
{ int[][][] a = new int [4] [3] [2];
                Scanner sc = new Scanner(System.in);
System.out.println ("Initializing the array ");

                for (int i = 0; i<a.length; i++){
                     for (int j = 0; j <a[0].length; j++) {
                          for (int k = 0; k <a[0][0].length; k++) {
  a[i][j][k] = sc.nextInt();
                          }
                     }
                }
System.out.println(" output ");
for (int i = 0; i<a.length; i++) {
for (int j = 0; j <a[0].length; j++) {
for (int k = 0; k <a[0][0].length; k++) {
System.out.print(a[i][j][k] + " ");
}
System.out.print("\t"); // space between matrices
}
System.out.println();
                }

        }
}
```

Output:

```
C:\nrcm>javac TestArray.java

C:\nrcm>java TestArray
Initializing the array
1 2
2 3
4 5

2 4
3 4
5 6

2 5
4 6
5 7

6 7
7 8
8 9
 output
1 2     2 4     2 5     6 7
2 3     3 4     4 6     7 8
4 5     5 6     5 7     8 9
```

Declare, Create and Initialize 2-dimensional & 3-dimensional arrays in one step. int[][] a = {{1,2,3}, {4,5,6}, {7,8}};
int[][][] b = {{{1,2}, {3,4}}, {{7,8,9}, {10}}};

2. Arrays of Object References:

a) Using classes

Syntax:

class_name[] ref_var = new class_name[size];

Program1: JavaProgram without using array of objects

```
class Student {
        int rno;
        float marks;
Student()
{ rno = 1;
              marks = 90.3f;
        }
Student(int sno, float marks) {
```

```
this.rno = sno;
this.marks = marks;
        }
        void display()
{ System.out.println("rno is " + rno);
System.out.println("marks are " + marks);
        }
        public static void main(String[] args)
                { Student s1 = new Student();
                Student s2 = new Student(2, 90.0f);
                Student s3 = new Student(3, 91.0f);
                Student s4 = new Student(60, 93.5f);
                s1.display();
                s2.display();
                s3.display();
                s4.display();
        }
}
```

Output:

```
C:\nrcm>javac Student.java

C:\nrcm>java Student
rno is 1
marks are 90.3
rno is 2
marks are 90.0
rno is 3
marks are 91.0
rno is 60
marks are 93.5
```

Program 2: Java Program using array of objects

```
class Student {
        int rno;
        float marks;
Student()
{ rno = 1;
                marks = 89.4f;
        }
Student(int r, float m)
{ rno = r;
```

```
            marks = m;
      }
      void display()
{ System.out.println(rno+"
"+marks);
      }
      public static void main (String[] args) {
            // declaration & initialization of an array
            Student [] s = new Student [4];
s[0] = new Student();
s[1] = new Student(2, 93.4f);
s[2] = new Student(3, 98.5f);
s[3] = new Student (4, 97.3f);
            s[0].display();
            s[1].display();
            s[2].display();
            s[3].display();
            for (int i = 0; i<s.length; i++)
                  { s[i].display();
System.out.println(s[i].rno + " " + s[i].marks);
            }
```

Output:

```
C:\nrcm>javac Student.java

C:\nrcm>java Student
1 89.4
2 93.4
3 98.5
4 97.3
1 89.4
1 89.4
2 93.4
2 93.4
3 98.5
3 98.5
4 97.3
4 97.3
```

2. Arrays of Object References:

b) Using interfaces

Syntax:

interface_name[ ] var =new interface_name[size];

```
 // 1. Define the interface
  Swimmer {
id swim();
  }
  // 2. Different classes implement the interface
  class Duck implements Swimmer {
 lic void swim() {
tem.out.println("The duck paddles through the water.");


 s Submarine implements Swimmer
 { lic void swim() {
tem.out.println("The submarine dives into the deep ocean.");


. Put them together in an array
ss Main {
blic static void main(String[] args) {
 eclare and initialize an array of the Swimmer interface
immer[ ] fleet = new Swimmer[2];
 ssign different implementing objects to the array elements
 t[0] = new Duck();
 t[1] = new Submarine();
oop through the array and invoke the interface method
(Swimmer s : fleet) {
wim(); // Polymorphism in action

 }
```

Output:

```
C:\nrcm>javac Main.java

C:\nrcm>java Main
The duck paddles through the water.
The submarine dives into the deep ocean.
```

Nested Classes:

- In Java, it is also possible to nest classes (a class within a class). The purpose of nested classes is to group classes that belong together, which makes your code more readable and maintainable
- To create an object of a non-static inner class, you must first create an object of its outer class. Then, use the outer class object to create the inner class object.

```
class OuterClass {
        int x = 10;
        class InnerClass {
                int y = 5;
        }
}
class InnerClassDemo {
         public static void main(String[]
args){ OuterClassmyOuter = new OuterClass();
OuterClass.InnerClassmyInner = myOuter.newInnerClass();
System.out.println(myInner.y + myOuter.x);
         }
}
```

Output:

```
C:\nrcm>javac InnerClassDemo.java

C:\nrcm>java InnerClassDemo
15
```

Strings

String:A String is a sequence of characters used to represent text. Strings can contain letters, numbers, symbols, and spaces. String is a predefined class in java.

They are usually enclosed in quotation marks: "Hello"

They are used to store information such as names, addresses, messages, passwords, and sentences.

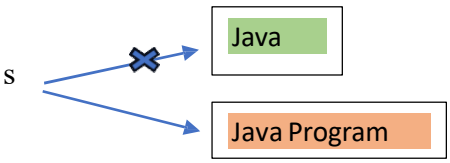
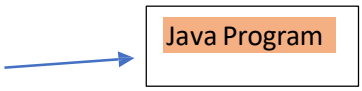
Strings-Immutability

In Java, String objects are immutable, meaning once a String object is created, its value cannot be changed.

- When you modify a String (e.g., using.concat() or .substring()), Java doesn't change the original object. Instead, it creates a brand-new String object in memory.

- Immutability provides security (e.g., for file paths, network connections), thread safety, and allows for the existence of the String Constant Pool.

Case (i): Immutable vs Mutable

Immutable	Mutable
String s=new String(“Java”); s.concat(“Program”); System.out.println(s);  once we created a string object, we are not allowed to perform any change in the existing object. If we are trying to perform any change with those changes a new object will be created. This behaviour is nothing but immutability.	StringBuffersb=new StringBuffer ("Java"); sb.append(“Program”); System.out.println(sb);  once we created a String Buffer object we can perform any change in existing object. This behaviour is nothing but mutability.

Case (ii):

Ex1:

String s1= new String ("Ramu");

String s2= new String ("Ramu");

System.out.println(s1= =s2); →false

System.out.println (b1.equals (s2)); →true

Ex2: String s1= "Ramu";

String s2= "Ramu";

System.out.println(s1= =s2); →true

System.out.println (b1.equals (s2)); →true

In String class equals () method is override for content comparison. Hence if content is same equals () method return true, even though objects are different.

StringBuffer s1=new StringBuffer(“SURYA”);

StringBuffer s2=new StringBuffer(“Surya”);

System.out.println(s1==s2); →false

System.out.println (b1. equals (s2)); →false

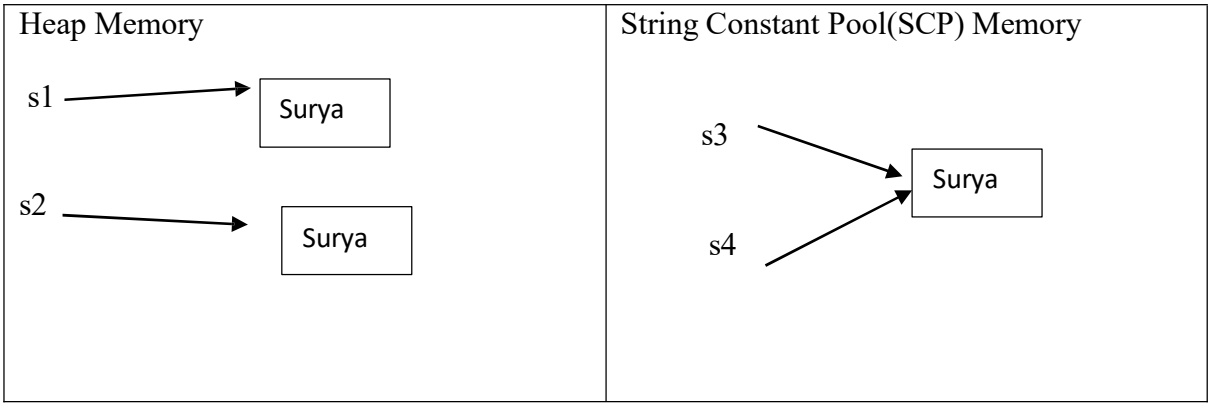
In StringBuffer.equals() method is not overridden for content comparison. Hence object class equals () method will be executed, which is always meant for reference comparison. Hence equals () method returns false, even their content is same if objects are different.

String Constant Pool (SCP):

1. Object creation in String Constant Pool (SCP) is always optional. First JVM will check if any abject already available with required content or not, if it in already available then it will reuse existing object Instead of creating new one. If object is not already available then only a new object will be created. Hence there is no chance of existing two objects in SCP with same content i.e; duplicate objects are not allowed in SCP.

2. Garbage collector gc can't access SCP area. Hence even though object doesn't have any reference, it is not eligible for garbage collection if it is present in SCP.
3. All SCP objects will be destroyed at the time of JVM shutdown.

Ex: String s1= new string ("Surya");
Sting s2= new string ("Surya");
String s3= "Surya";
String s4= "Surya";



Constructors of String class:

1. **String s=new String ();**

Creates an empty string object

2. **String s =new String (String s);**

Ex: String = new String ("Surya");

Methods of String class:

1. **char charAt (int index):** returns the character locating at specified index

Ex: String = "Ramu";

System.out.println (s.charAt (2)); → m

System.out.println (s. charAt(20));

R.E: "StringIndexOutOfBoundsException"

2. **String concat (String s):**

Ex: String s1 = "Rama";

s1 = s1.oncat ("raju");

//s1 = s1+"raju"

System.out.println (s1); → Ramaraju

3. **boolean equals (Object o):** For checking content comparison where case is also important.
4. **booleanequalsIgnoreCase(String s):** For checking content comparison where case is not important.

   Ex: String s= "JAVA",

   System.out.println(s.equals ("java"));→false.

   System.out.prinltn (s.equasIgnoreCase ("java")); → true.

5. **String subString (int begin):**

returns the substring from begin index to end of the String.

6. **String subSting (int begin, int end):**

returns the substring from 'begin' index to 'end-1' index.

Ex:String=" abcdefgh";

System.out.println (s.substring (2)); →cdefgh

System.out.println (s.substring (2,5)); →cde

7. **int length():**

returns the number of characters present in the string.

Ex:String s = "Surya";

System.out.println (s.length); →CE: cannot find symbol

System.out.println(s.length()); →5

8. **String replace (char old, char new):**

Ex: String s= "abababb"

System.out.println (s.replace('a', "b")); →bbbbbb

9. **String toLowerCase():**
10. **String toUpperCase():**

Ex: String s1= "krishna";

String s2= s1.toUpperCase();

String s3= s1. toLowerCase();

System.out.println (s1==s2); →false.

System.out.println (s1= =s3); → true.

11. **String trim():** To remove the blank spaces present at beginning and end of the string but not middle blank space

12. **int indexOf (char ch):** It returns the index of first occurrence of specified character, if the character is not available then returns '-'

13. int lastIndexOf(char ch): It returns the index of last occurrence of specified Character, if the character is not available then returns '-'

Once we created a String object, we are not allowed to perform any change in the existing object. If we are trying to perform any changes, with those changes a new object will be created. Because of our operation if there is no change in the content then it is not required to create any new object.

Eg :String s1= "Surya";
String s2 = s1. toString ();
System.out.println (s1 = =s2); →true

Program1: Java Program to demonstrate methods of the String class.

```
class StringBasicsDemo {
        public static void main(String[] args)
                { String s1 = "Rama";
                String s2 = "Krishna";
        System.out.println(s1.length()+"  "+s2.length());
        System.out.println(s1.charAt(0)+" "+s2.charAt(3));
        System.out.println(s1.toUpperCase()+" "+s2.toLowerCase());
        System.out.println(s1+" "+s2);
        System.out.println(s1.concat(s2));
        System.out.println(s1.equals("rama"));
        System.out.println(s1.equalsIgnoreCase("rama"));
                String com= "   " + s1 + " " + s2 + " " + s1 + "   ";
                String trimmed=com.trim();
        System.out.println(trimmed);
        System.out.println(trimmed.replace("Rama", "Radha"));
        System.out.println(s1.indexOf('R'));
        System.out.println(s1.lastIndexOf('a'));
        System.out.println(s2.substring(1, 4));
        }
}
```

Output:

```
C:\nrcm>javac StringBasicsDemo.java

C:\nrcm>java StringBasicsDemo
4  7
R s
RAMA krishna
Rama Krishna
RamaKrishna
false
true
Rama Krishna Rama
Radha Krishna Radha
0
3
ris
```

Program2: Java program that checks whether a given string is a palindrome or not.

```
import java.util.Scanner;
import java.lang.*;
class PalindromeCheck {
        public static void main(String[] args)
                { Scanner sc = new
                Scanner(System.in);
        System.out.print("Enter a string: ");
                String original = sc.nextLine();
                String reversed = "";

                 // Reverse the string
                 for (int i = original.length() - 1; i>= 0; i--)
                       { reversed = reversed+ original.charAt(i);
        }
        // Check palindrome
        if (original.equalsIgnoreCase(reversed))
        { System.out.println(original + " is a palindrome.");
        }
        else {
        System.out.println(original + " is not a palindrome.");
        }
        sc.close();
        }
  }
```

Output:

```
C:\nrcm>javac PalindromeCheck.java

C:\nrcm>java PalindromeCheck
Enter a string: madam
madam is a palindrome.
```

Program 3: Java program for sorting a given list of names in ascending order

```
import java.util.Scanner;
import java.util.Arrays;
class SortNames {
        public static void main(String[] args)
                { Scanner sc = new Scanner(System.in);
System.out.print("Enter number of names: ");
                int n = sc.nextInt();
sc.nextLine(); // consume newline
String[] names = new String[n];
System.out.println("Enter the names:");
                for (int i = 0; i< n; i++)
                { names[i] = sc.nextLine();
                }
                // Sort Names
Arrays.sort(names);
System.out.println("Names in ascending order:");
                for (String name : names) {
System.out.println(name);
                }
sc.close();
        }
}
```

Output:

```
C:\nrcm>javac SortNames.java

C:\nrcm>java SortNames
Enter number of names: 3
Enter the names:
Anu
Anil
Anuradha
Names in ascending order:
Anil
Anu
Anuradha
```

StringBuffer:

If the content is changing frequently then it is never recommended to use String concept, because for every change a new object will be created.

To handle this type of requirement, compulsory we should go for **StringBuffer** where all changes will be performed in the existing object only.

Constructorsof StringBuffer class

1) **StringBuffer sb = new StringBuffer();**

Creates an empty StringBuffer object with default initial capacity **'16'**.
Once StringBuffer reaches its maximum capacity then a new StringBuffer object will be created with **new capacity = ({current capacity} + 1) * 2**

2) **StringBuffer sb =new StringBuffer (int initialcapacity**):
creates an empty StringBuffer object with the specified initial capacity

3) **StringBuffer sb = new StringBuffer(String s);**
Creates an equivalent StringBuffer for the given String object with
capacity = {s.length()} + 16

Ex: StringBuffer sb = new StringBuffer("Surya");
System.out.println (sb.capacity()); → 21

Methods of StringBuffer class

1) int length()
2) int capacity()
3) char charAt(int index):
Ex: StringBuffer sb = new StringBuffer("Surya");
System.out.println (sb.charAt(2)); → r
System.out.println (sb.charAt(20));
R.E: "StringIndexOutOfBoundsException"
4) void setCharAt(int index, char ch):
To replace the character present at specified index with the provided character.
5) StringBufferappend(String s);
Ex: StringBuffer sb = new StringBuffer();
sb.append("Surya");
System.out.println(sb);→ Surya

6) **StringBufferinsert(int begin, String s)**
To insert at a specified index, use this method.
Ex: StringBuffer sb = new StringBuffer("Java");
sb.insert(5, "Program");
System.out.println(sb); → Java Program
7)StringBufferreverse():
Ex: StringBuffer sb = new StringBuffer("Surya");
System.out.println (sb.reverse());
Program4:: Java Program to demonstrate methods of the StringBuffer class

```
class StringBufferExample {
```

```
    public static void main(String[] args) {
        // Create a StringBuffer object
StringBuffersb = new StringBuffer("Hello");
sb.append(" World");
System.out.println("After append: " + sb);
sb.insert(5, " Java");
System.out.println("After insert: " + sb);
sb.replace(6, 10, "C++");
System.out.println("After replace: " + sb);
sb.delete(5, 9);
System.out.println("After delete: " + sb);
sb.reverse();
System.out.println("After reverse: " + sb);
System.out.println("Length: " + sb.length());
System.out.println("Character at index 2: " + sb.charAt(2));
    }
}
```

Output:

```
C:\nrcm>javac StringBufferExample.java

C:\nrcm>java StringBufferExample
After append: Hello World
After insert: Hello Java World
After replace: Hello C++ World
After delete: Hello World
After reverse: dlroW olleH
Length: 11
Character at index 2: r
```

StringTokenizer:

The StringTokenizer class in Java is used to break a string into smaller parts called tokens based on specified delimiters. It maintains an internal position to track the next token and allows sequential extraction of tokens from a string.

- Supports single or multiple delimiters.
- Implements the Enumeration interface.
- Provides methods to traverse tokens one by one.

Constructors of StringTokenizer class:

1. **StringTokenizer (String str):** Creates a tokenizer for the specified string. Uses default delimiters (whitespace, tabs, etc.)

2. StringTokenizer (String str, String delim): Creates a tokenizer for the specified string using the given delimiters.

3. StringTokenizer(String str, String delim, booleanreturnDelims):is used to create a StringTokenizer with a **custom delimiter** and an option to **return delimiters as tokens**.

returnDelims

- true → Delimiters are also returned as tokens.
- false → Delimiters are ignored (default behavior).

Ex: Constructor3

a) returnDelims = false

```
StringTokenizer st = new StringTokenizer("Apple,Banana,Mango", ",", false);

 while (st.hasMoreTokens())
{ System.out.println(st.nextToken());
}
```

Output: Apple
Banana
Mango

b) returnDelims = true

```
StringTokenizer st = new StringTokenizer("Apple, Banana, Mango", ",", true);

            while (st.hasMoreTokens())
{ System.out.println(st.nextToken());
  }
```

Output: Apple,
Banana,
Mango

Methods of StringTokenizerclass

1. **countTokens()**: Returns the total number of tokens present.
2. **hasMoreTokens():** Tests if tokens are present for the StringTokenizer's string.
3. **nextElement():** Returns an Object rather than String.
4. **hasMoreElements():** Returns the same value as hasMoreTokens().
5. **nextToken():** Returns the next token from the given StringTokenizer.

Program5: Java Program to demonstrateStringTokenizer class

```
import java.util.*;

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

        // Creating a StringTokenizer
StringTokenizer st = new StringTokenizer("Welcome to Java Programming ");
StringTokenizer st1 = new StringTokenizer("");

        // countTokens Method
        int c = st.countTokens();
System.out.println(c);

        // hasMoreTokens Method
System.out.println("Welcome to Java Programming: " + st.hasMoreTokens());
System.out.println("(Empty String): " + st1.hasMoreTokens());

        // nextElement() Method
System.out.println("\nTraversing the String:");

        while (st.hasMoreTokens())
{ System.out.println(st.nextElement());
        }
    }
}
```

Output:

```
C:\nrcm>javac StringTokenizerDemo.java

C:\nrcm>java StringTokenizerDemo
4
Welcome to Java Programming: true
(Empty String) : false

Traversing the String:
Welcome
to
Java
Programming
```

Program2: Java Program that reads a line of integersand then displays each integer and the sum of all the integers (Use StringTokenizer class of java.util)

```
import java.util.Scanner;
import java.util.StringTokenizer;
```

```
class IntegerTokenizer {
        public static void main(String[] args)
            { Scanner sc = new Scanner(System.in);
System.out.print("Enter integers separated by space: ");
            String input = sc.nextLine();
StringTokenizer st = new StringTokenizer(input);
            int sum = 0;
System.out.println("Entered integers:");
            while (st.hasMoreTokens()) {
            int num = Integer.parseInt(st.nextToken());
System.out.println(num);
            sum += num;
            }
System.out.println("Sum of integers: " + sum);
sc.close();
        }
}
```

Output:

```
C:\nrcm>javac IntegerTokenizer.java

C:\nrcm>java IntegerTokenizer
Enter integers separated by space: 2 3 4
Entered integers:
2
3
4
Sum of integers: 9
```

Inheritance in Java:

Inheritance is a fundamental feature of Java that allows one class to inherit the variables and methods of another class. It is implemented using the **extends** keyword

The process of getting variables and methods of one class into another class is called inheritance.

Inheritance represents **is-a** relationship, that means a subclass is a specialized form of its superclass. For example, if Dog extends Animal, then a Dog **is-a**Animal.

All the variables and methods of super class are inherited to its subclass, but constructors of the superclass are not inherited to its subclass

Advantages of Inheritance

- Code Reusability
- Method Overriding

- Runtime Polymorphism
- Extensibility of Applications

Types of inheritance in Java:

Java supports the following 3 types of inheritance

1. Single level inheritance
2. Multi level inheritance
3. Hierarchical inheritance

Java does not support the following 2 types of inheritance

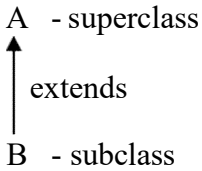
4. Multiple inheritance
5. Hierarchical inheritance

1. Single level inheritance:

In single level inheritance, a **subclass** (derived class) inherits the variables and methods of a **single superclass** (base class) using the **extends** keyword.

Syntax:**class B extends A**, here A is superclass and B is subclass

Inheritance Structure:



Program1:Single level inheritance

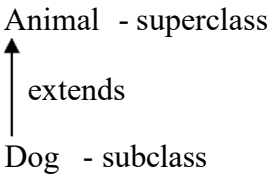
```
// Single level inheritance
class A {
   void display()
{ System.out.println("Class A");
   }
}
class B extends A
   { void show() {
System.out.println("Class B");
   }
    public static void main(String[] args)

{ B b1 = new B();

            b1.display();
```

```
        b1.show();
    }
}
```

```
C:\nrcm>javac B.java

C:\nrcm>java B
Class A
Class B
```

Program2: Single level inheritance

Inheritance Structure:



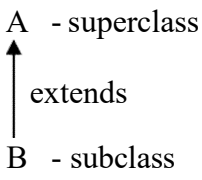

```
// Superclass (Base Class)
class Animal {
void eat() {
System.out.println("The animal is eating food.");
}
}
// Subclass (Derived Class) inheriting from Animal
class Dog extends Animal {
void bark() {
System.out.println("The dog is barking.");
}
}
// Main Class to execute the program
 class Test {
public static void main(String[] args) {
// Create an object of the subclass
Dog myDog = new Dog();
// Call method from the Superclass (inherited)
myDog.eat();
// Call method from the Subclass
```

```
myDog.bark();
}
}
```

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
The animal is eating food.
The dog is barking.
```

Program3: Single level inheritance

Inheritance Structure:




```
class A {
    int a = 10;
    void m1(){
System.out.println("A's m1() method");
    }
}
class B extends A
    { int b = 20;
    void
m2(){ System.out.println("B's m2()
method");
    }
}
class Test {
    public static void main(String[] args)
{ B b1 = new B();
System.out.println(b1.a + " " + b1.b);
 b1.m1();
b1.m2();
}
```

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
10 20
A's m1() method
B's m2() method
```

2. Multilevel Inheritance

In multilevel inheritance, a class inherits from a subclass, creating a chain of inheritance (e.g., Class C inherits from Class B, and Class B inherits from Class A).

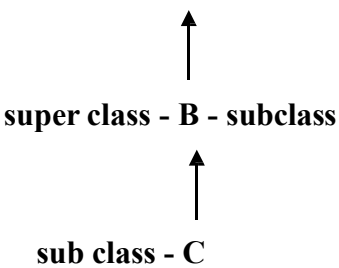
```
class A {

}

class B extends A {

}

class C extends B {

}
```

Diagram:

Program 1: Multi level inheritance

Inheritance Structure:

A - super class



```
class A
 { int a;
void m1() {
System.out.println("A's method is m1()");
 }
A(int a)
{ this.a = a;
}
}
class B extends A
{ int b;
void m2() {
System.out.println("B's method is m2()");
}
B(int a, int b) {
super(a);
this.b = b;
}
}
class C extends B
{ int c;
 void m3() {
System.out.println("C's method is m3()");
}
C(int a, int b, int c)
{ super(a, b);
this.c = c;
}
}
class Test {
public static void main(String[] args) {
 A a1 = new A(10);
 B b1 = new B(20, 30);
C c1 = new C(40, 50, 60);
System.out.println(a1.a);
 a1.m1();
System.out.println(b1.a + " " + b1.b);
 b1.m1();
b1.m2();
System.out.println(c1.a + " " + c1.b + " " + c1.c);
c1.m1();
c1.m2();
```

```
c1.m3();
} }
```

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
10
A's method is m1()
20 30
A's method is m1()
B's method is m2()
40 50 60
A's method is m1()
B's method is m2()
C's method is m3()
```

Program 2: Multi level inheritance

Inheritance Structure:

Animal - super class

super class - Dog - subclass

sub class - Puppy

```
// superclass
class Animal {
   void eat() {
System.out.println("This animal eats food.");
   }
}
// Subclass (Intermediate Class) inheriting from Animal
class Dog extends Animal {
   void bark()
{ System.out.println("The dog
barks.");
```

```
}
// Subclass (Child Class) inheriting from Dog
class Puppy extends Dog {
    void weep() {
System.out.println("The puppy is weeping.");
    }
}

// Main Class to execute the program
public class Main {
    public static void main(String[] args) {
        // Create an object of the lowest subclass
        Puppy myPuppy = new Puppy();

        // Accessing methods from all levels of the inheritance chain
myPuppy.eat();  // Inherited from Animal
myPuppy.bark(); // Inherited from Dog
myPuppy.weep(); // Defined in Puppy
    }
}
```

```
C:\nrcm>javac Main.java

C:\nrcm>java Main
This animal eats food.
The dog barks.
The puppy is weeping.
```

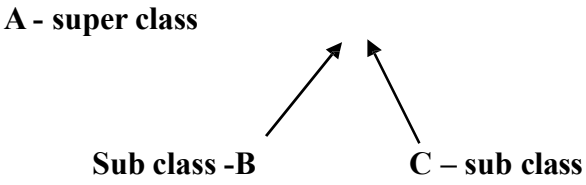
3. Hierarchical Inheritance

Multiple child classes inherit from the same parent class.

Syntax:

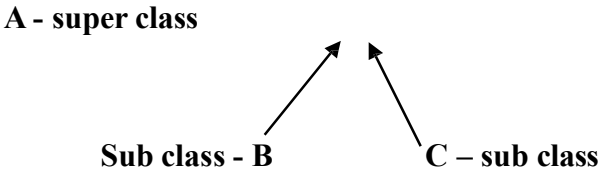
```
class A {

}

class B extends A {

}

class C extends A {
```

}



Program 1: Hierarchical inheritance

Inheritance structure

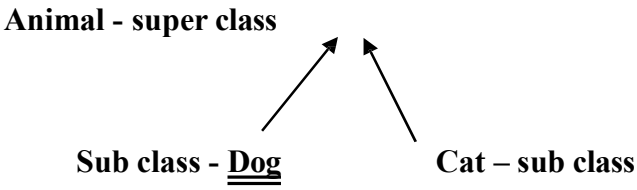



```
  class A {
        int a = 10;
 void m1() {
System.out.println("A's m1() method");
}
}
class B extends A
{ int b = 20;
 void m2() {
System.out.println("B's m2() method");
}
}
class C extends A
{ int c = 30;
 void m3() {
System.out.println("C's m3() method");
}
}
class Test {
public static void main(String[] args)
{ A a1 = new A();
System.out.println(a1.a);
a1.m1();
B b1 = new B();
System.out.println(b1.a + " " + b1.b);
b1.m1();
b1.m2();
```

```
 C c1 = new C();
System.out.println(c1.a + " " + c1.c);
c1.m1();
c1.m3();
}
}
```

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
10
A's m1() method
10 20
A's m1() method
B's m2() method
10 30
A's m1() method
C's m3() method
```

Inheritance Structure:

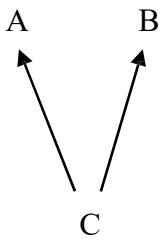



```
  // superclass
  class Animal {
        void eat() {
System.out.println("The animal is eating food.");
        }
  }
  // subclass 1 (inherits from Animal)
  class Dog extends Animal {
        void bark() {
System.out.println("The dog is barking: Woof Woof!");
        }
  }
  // subclass 2 (Inherits from Animal - making it Hierarchical)
  class Cat extends Animal {
```

```
        void meow() {
System.out.println("The cat is meowing: Meow Meow!");
        }
  }
    class Main {
        public static void main(String[] args)
{ System.out.println("--- Dog Behavior ---");
                Dog myDog = new Dog();
myDog.eat();  // Inherited method
myDog.bark(); // Specific method
System.out.println("\n--- Cat Behavior ---");
Cat myCat = new Cat();
myCat.eat();  // Inherited method
myCat.meow(); // Specific method
}
}
```

```
C:\nrcm>javac Main.java

C:\nrcm>java Main
--- Dog Behavior ---
The animal is eating food.
The dog is barking: Woof Woof!

--- Cat Behavior ---
The animal is eating food.
The cat is meowing: Meow Meow!
```

4. Multiple Inheritance: A class inherits from more than one parent class



Java does not support multiple inheritance through classes because it can create ambiguity problems. However, Java supports multiple inheritance through interfaces.

Example1:

// Compiler Error

```
class A{
 ……..
```

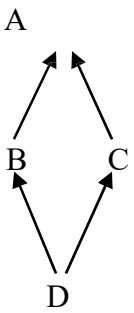
```
}
Class B {
 ……..
}
Class C extends A,B {
……..
}
```

Example 2:

```
interface A
{ void
methodA();
}
interface B
{ void
methodB();
}
class C implements A, B
{ public void methodA()
{ System.out.println("Method
A");
}
public void methodB()
{ System.out.println("Method B");
}
}
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance.



Java does not support hybrid inheritance through classes because it includes multiple inheritance.

Summary table: inheritance

Type	Description	Supported in Java
Single level Inheritance	One parent, one child	Yes

Multilevel Inheritance	Chain of inheritance	Yes
Hierarchical Inheritance	One parent, many children	Yes
Multiple Inheritance	Multiple parents, one child	No (Classes), Yes (Interfaces)
Hybrid Inheritance	Combination of inheritance types	No (Classes), Yes (Interfaces)

super keyword:

The super keyword is used to refer the members (variables, methods and constructors) of the immediate superclass from the subclass.

Different ways to use super keyword:

1. To access superclassvariables

When a superclass and subclass are having the variablewith the same name, thenuse super to access the superclass variable from the subclass.

Syntax:super.variableName;

Program1: Java program to access superclass variables from subclass

```
class Animal {
   String color = "White";
}
class Dog extends Animal
   { String color = "Black";
   void display() {
System.out.println(color);        // subclass variable
System.out.println(super.color); // superclass variable
   }
   public static void main(String[] args)
      { Dog d = new Dog();
d.display();
   }
}
```

Output:

```
C:\nrcm>javac Dog.java

C:\nrcm>java Dog
Black
White
```

2. To access superclass methods

When superclass and subclass are having the method with the same prototype, then use super to access the superclass method from the subclass.

. **Syntax**: super.methodName();

Program2: Java program to access superclass method from subclass

```
class Animal
   { void sound() {
System.out.println("Animal makes a sound");
   }
}
class Dog extends Animal
   { void sound() {
System.out.println("Dog barks");
   }
  void display() {
super.sound(); // Calls parent method
sound();       // Calls child method
   }
   public static void main(String[] args)
      { Dog d = new Dog();
d.display();
   }
}
```

Output:

```
C:\nrcm>javac Dog.java

C:\nrcm>java Dog
Animal makes a sound
Dog barks
```

3. To invoke superclass constructors

a) super( )

b) super(parameters)

3 a) super( )

i) **super()** is used to call the no-argument constructor of the superclass. It must be the first statement in a subclass constructor. If **super()** is not written explicitly, the Java compiler automatically inserts it only if the superclass has a no-argument constructor.

Syntax:super();

Program3: Java program using super( )

```
class Animal
{ Animal() {
System.out.println("Animal Constructor");
   }
  }
class Dog extends Animal
{ Dog() {
super(); // calls superclass constructor
System.out.println("Dog Constructor");
   }
   public static void main(String[] args)
      { Dog d = new Dog();
   }
}
```

Output:

```
C:\nrcm>javac Dog.java

C:\nrcm>java Dog
Animal Constructor
Dog Constructor
```

ii) Constructor chaining using super()

Constructor chaining means calling one constructor from another constructor.
In inheritance, the subclass constructor can call the superclass constructor using super().Inside each constructor (whether user-defined or compiler-generated), the first statement must be eithersuper() orthis().

Program4: Java program on Constructor Chaining

```
class A
{ A(){
System.out.println("A's constructor");
   }
}
class B extends A
{ B(){
System.out.println("B's constructor");
   }
```

```
}
class C extends B
{ C(){
System.out.println("C's constructor");
    }
}
class Test{
  public static void main(String
  args[]){ C obj=new C();
  }
}
```

Output:

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
A's constructor
B's constructor
C's constructor
```

3 b) super(parameters)

super(parameters) is used to call a parameterized constructor of the superclass. It must be the first statement in a subclass constructor. It passes values from the subclass constructor to the superclass constructor. t is required when the superclass has only parameterized constructors or when a specific superclass constructor needs to be invoked.

Program 5: Java program on super(parameters)

```
class Box {
          int length, width, depth;
Box(int length, int width, int depth)
{ this.length= length;
this.width = width;
this.depth= depth;
          }
         void volume() {
System.out.println("Volume is = " +(length * width * depth));
            }
    }
class BoxWeight extends Box
   { int weight;
BoxWeight(int l, int w, int d, int wt)
{ super(l, w, d);  // Calls Box constructor
```

```
    weight = wt;
    }
}
class SuperDemo {
    public static void main(String[] args)
{ BoxWeight b1 = new BoxWeight(10, 20, 30, 40);
BoxWeight b2 = new BoxWeight(1, 2, 3, 4);
    b1.volume();
    b2.volume();
System.out.println("Weight of b1 = " + b1.weight);
System.out.println("Weight of b2 = " + b2.weight);
    }
}
```

Output:

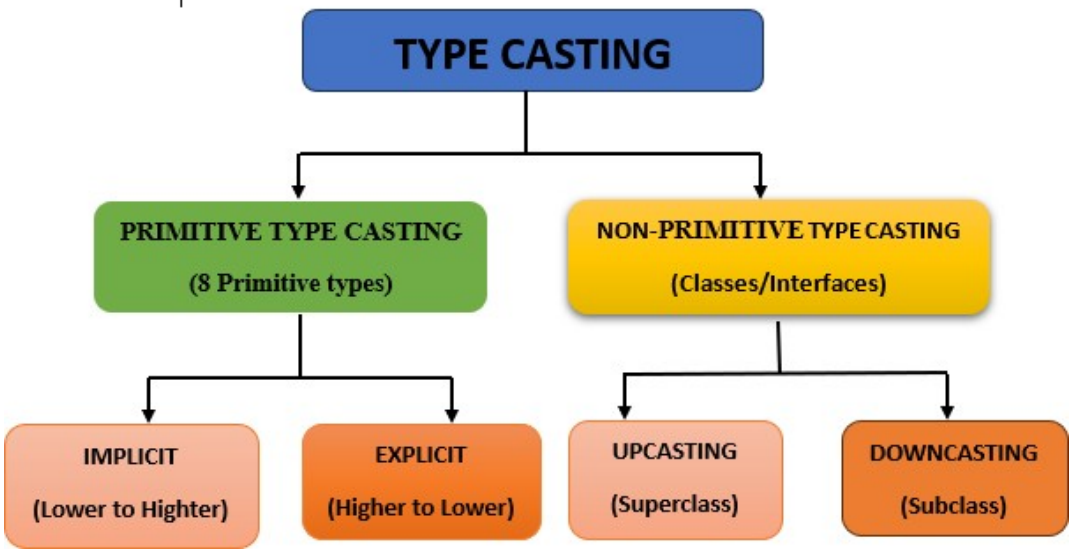
```
C:\nrcm>javac SuperDemo.java

C:\nrcm>java SuperDemo
Volume is = 6000
Volume is = 6
Weight of b1 = 40
Weight of b2 = 4
```

Summary table on super keyword

Syntax	Purpose
super ()	Calls superclass default constructor
super(parameters)	Calls superclass parameterized constructor
super.variable	Access superclass variable
super.method	Call superclass method

Type Casting

Type casting is the process of converting a value of one data type into another data type.



1. Primitive Type Casting

- **Implicit Type Casting**: Happens automatically when a smaller data type value is assigned to a larger data type variable.
- **Explicit Type Casting:** Required when assigning a larger data type value to a smaller data type variable. This requires manual casting and can lead to data loss.

Program 1: Java program on Primitive Type Casting

```
class Test {
    public static void main(String[] args) {
        // Implicit Type Casting (Done by the Compiler)
        byte b = 10;
        int a = b; // Lower (byte) assigned to Higher (int) automatically
        // Alternatively: int a = (int) b;
System.out.println(a + b);
System.out.println(a + " " + b);
        // Explicit Type Casting(Done by the Programmer)
        int x = 100;
        // byte y = x; // Compiler Error
        byte y = (byte) x; // Explicit cast to narrow the value
System.out.println(x + " " + y);
    }
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
20
10 10
100 100
```

Non-Primitive Type Casting

Non-primitive type casting is done with reference variables (classes/interfaces) and requires an inheritance relationship between the classes.

A. Upcasting

- **Definition**: Assigning a subclass object to a superclass reference variable.
- **Syntax:**Super_Classref_var = new Sub_Class();// casting is done by compiler
  Or
  Super_Classref_var = (Super_Class) new Sub_Class();//by programmer
- **Access Rule**: With the superclass reference variable, we can access all members of the superclass except overridden methods. In case of overridden methods of the superclass, subclass overriding methods are executed.

B. Downcasting

- **Definition:** Assigning a superclass reference to a subclass reference variable.
- **Rule:**Downcasting must be followed by upcasting. It cannot be done directly on a raw superclass object, it will throw a runtime error.
- **Syntax:**Super_Classref_var = new Sub_Class(); //upcasting
  Sub_Class ref_var1=(Sub_Class) ref_var;//downcasting
- Why Use It? To access the specific members of the subclass that are hidden during upcasting.

```
class Animal {
   int a = 10, b=20;
   //overridden method
   void eat() {
System.out.println("Animal's eat() method");
   }
   void sleep() {
System.out.println("Animal's sleep() method");
   }
}
class Dog extends Animal
   { int c=30;
```

```
    // Overriding method
    void eat() {
System.out.println("Dog's eat() method");
    }
    void bark() {
System.out.println("Dog's bark() method");
    }
}
class Test {
    public static void main(String[] args) {
        //upcasting
        Animal a1 = new Dog(); // Animal a1 =(Animal) new Dog();
System.out.println(a1.a + " " + a1.b);
a1.eat();  //Dog's eat() method
        a1.sleep();
        //downcasting
        Animal a2 = new Dog();  //upcasting
        Dog d1 = (Dog) a2; //downcasting
System.out.println(d1.a + " " + d1.b + " " + d1.c);
d1.eat();
        d1.sleep();
        d1.bark();
    }
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
10 20
Dog's eat() method
Animal's sleep() method
10 20 30
Dog's eat() method
Animal's sleep() method
Dog's bark() method
```

Object Instantiation Rules and Exceptions:

When working with reference variables under inheritance, let's look at the behaviour of different assignment strategies:

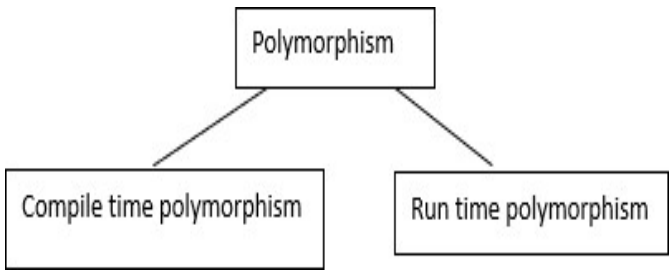
1. Animal a = new Animal();
    - Valid. Pure Superclass object.
2. Dog d = new Dog();

- Valid. Pure Subclass object.

3. Animal a = new Dog();
   - Valid. Upcasting.
4. Dog d = new Animal();
   - Invalid. Compile-time Error.
     A subclass reference cannot hold a superclass object directly.
5. Animal a = (Animal) new Dog();
   Dog d = (Dog) a;
   - Valid. Valid downcasting
6. Animal a = new Animal();
   Dog d = (Dog) a;
   - Compiles successfully but throws a runtime exception:
     - java.lang.ClassCastException

Polymorphism:

Polymorphism: **Polymorphism** is one of the fundamental principles of Object-Oriented Programming (OOP).The word **polymorphism** means **"many forms"**. It allows the same method or object reference to perform different actions depending on the situation. Polymorphism increases code flexibility, reusability, and maintainability. In Java, it enables a superclass reference to refer to objects of different subclasses.



Compile time polymorphism	Run time polymorphism
Polymorphism occurs during compile time.	Polymorphism occurs during runtime.
Ex: Overloading (Method overloading, Constructor overloading)	Ex: Overriding (Method overriding)

1.Compile time polymorphism

a) Constructor Overloading:

If two or more constructors having the same name and they may differ either in the number

of arguments or type of arguments or order of arguments then the constructors are said to be overloaded constructors.

Example 1:

```
class Student
{ Student() {
              .......
        }
Student(int rno) {
              .......
        }
  }
```

In this case, the constructor Student() is said to be overloaded constructor because they are differing by the **number** of arguments.

Example 2:

```
class Student {
Student(int no, float marks) {
     ……..
          }
Student(float marks, int no) {
        ……..
         }
}
```

In this case, the constructor Student() is said to be overloaded constructor because they are differing by the **order** of arguments.

Example 3:

```
class Student {
Student(int no, float marks) {
          ……..
            }
Student(float marks, String name) {
           ……..
            }
}
```

In this case, the constructor Student() is said to be overloaded constructor because they are differing by the **type** of arguments.

Program1: Java program on Constructor Overloading

```
class Student
   { int rno;
   float marks;
```

```
    String name;
   // User-defined no-arg constructor
Student() {
rno = 1;
        marks = 95.6f;
        name = "Arjun";
    }
    // User-defined parametrized constructor
Student(int no, float mark, String name)
{ this.rno = no;
this.marks = mark;
        this.name = name;
    }
    // Copy Constructor (User-defined parametrized constructor)
Student(Student obj) {
rno = obj.rno;
        marks = obj.marks;
        name = obj.name;
    }

void display()
{ System.out.println("rno is " + rno);
System.out.println("marks are " + marks);
System.out.println("name is " + name);
    }
}

class Demo {
    public static void main(String[] args)
        { Student S1 = new Student();
        Student S2 = new Student(2, 98.6f, "Abhimanyu");
        Student S3 = new Student(S1);
        S1.display();
        S2.display();
        S3.display();
    }
}
```

Output:

```
C:\nrcm>javac Demo.java

C:\nrcm>java Demo
rno is 1
marks are 95.6
name is Arjun
rno is 2
marks are 98.6
name is Abhimanyu
rno is 1
marks are 95.6
name is Arjun
```

b). Method Overloading:

If two or more methods having the same name and they may differ either in the number of arguments or type of arguments or order of arguments but not on the return type then the methods are said to be overloaded methods.

Example1:

```
class Test {
void sum(int a, float b) {
        ……..
}
int sum(float a, int b) {
        …….
}
}
```

In this case, the method sum() is said to be**overloaded** method because they are differing by the **order** of arguments. Return type may or may not be same.

Example 2:

```
class Test {
        void sum(int a) {
 ……..
        }
        int sum(int a) {
.........
        }
}
```

In this case, the method sum() is **not overloaded** method because they are differing by the return type but not on the number of arguments.

Program 1: Java Program on Method Overloading with sum() method

```
class OverloadDemo {
        void sum(int a, int b)
        { int c = a + b;
System.out.println("sum is " + (a + b));
        // Note from script: can be written as System.out.println(c);
         }
        int sum(int a, int b, int c)
        { int d = a + b + c;
        return d;
        }
        int sum(int a, int b, int c, int d)
        { return (a + b + c + d);
       }
}
class Test {
        public static void main(String[] args)
{ overloadDemo o1 = new overloadDemo();
o1.sum(10, 20);

            int x = o1.sum(10, 20, 30);
System.out.println(x);

            int y = o1.sum(10, 20, 30, 40);
System.out.println(o1.sum(10, 20, 30, 40));
        }
}
```

Output:

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
sum is 30
60
100
```

Program 2: Java Program on Method Overloading with area() method

```
class Area {
```

```
// Area of Square
void area(double side)
 { double a = side * side;
System.out.println("Area of Square = " + a);
}
   // Area of Rectangle
void area(double length, double breadth)
{ double a = length * breadth;
System.out.println("Area of Rectangle = " + a);
}
   // Area of Triangle
void area(intbase, intheight)
{ double a = 0.5 * base * height;
System.out.println("Area of Triangle = " + a);
}
}
class MethodOverloadingArea
{ public static void main(String[]args)
{
   Area obj = new Area();
obj.area(5);          // Square
obj.area(6, 4);       // Rectangle
obj.area(8, 5);    // Triangle
  }
```

Output:

```
c:\nrcm>javac MethodOverloadingArea.java

c:\nrcm>java MethodOverloadingArea
Area of Square = 25.0
Area of Rectangle = 24.0
Area of Triangle = 20.0
```

Program 3: Java Program on Method Overloading with volume() method

```
class Volume {
    // Volume of Cube
void volume(double side)
{ double v = side * side * side;
System.out.println("Volume of Cube = " + v);
}
        // Volume of Cylinder
        void volume(double radius, double height) {
```

```
                double v = 3.14159 * radius * radius * height;
System.out.println("Volume of Cylinder = " + v);
        }
  // Volume of Cuboid
void volume(double length, double breadth, double height)
{ double v = length * breadth * height;
System.out.println("Volume of Cuboid = " + v);
}
}
class MethodOverloadingDemo
{ public static void main(String[] args)
{
      Volume obj = new Volume();
obj.volume(5);          // Cube
obj.volume(4, 6);       // Cylinder
obj.volume(4, 5, 6);    // Cuboid
}
```

Output:

```
c:\nrcm>javac MethodOverloadingDemo.java

c:\nrcm>java MethodOverloadingDemo
Volume of Cube = 125.0
Volume of Cylinder = 301.59263999999996
Volume of Cuboid = 120.0
```

2. Runtime Polymorphism

Method overriding: If both superclass and subclass are having the method with same name and same prototype then subclass method will override the superclass method. In this case subclass method is called overriding method and superclass method is called overridden method.

Instance variable hiding: If both superclass and subclass are having the same variable then subclass variable will hide the superclass variable.

Program 1: Java Program on Method Overriding & Instance Variable Hiding

```
class Animal {
int a=10; //a is instance variable
//overridden eat() method
   void eat() {
System.out.println("Animal's eat() method ");
   }
```

```
    void sleep() {
System.out.println("Animal's sleep() method ");
    }
}
class Dog extends Animal {
    int a= 20; // a is instance variable
    // Overriding eat() method
    void eat() {
System.out.println("Dog's eat() method ");
    }

    void bark() {
System.out.println("Dog's bark() method");
    }
}
class MethodOverridingDemo {
    public static void main(String[] args)
{ Animal a1 = new Animal();
System.out.println(a1.a);
a1.eat();
a1.sleep();
          //method overriding
Dog d1= new Dog();
System.out.println(d1.a);
d1.eat();
d1.sleep();
d1.bark();
// Runtime polymorphism/upcasting
Animal obj = new Dog();
obj.eat();
    }
}
```

Output:

```
c:\nrcm>javac MethodOverridingDemo.java

c:\nrcm>java MethodOverridingDemo
10
Animal's eat() method
Animal's sleep() method
20
Dog's eat() method
Animal's sleep() method
Dog's bark() method
Dog's eat() method
```

Program 2: Java Program on Method Overriding & Instance Variable Hiding

```
class Animal {
     int a=10; //a is instance variable
    //overridden eat() method
    void eat() {
System.out.println("Animal's eat() method ");
    }
    void sleep() {
System.out.println("Animal's sleep() method ");
    }
}
class Dog extends Animal {
    int a= 20; // a is instance variable
    // Overriding eat() method
    void eat() {
System.out.println("Dog's eat() method ");
    }

    void bark() {
System.out.println("Dog's bark() method");
    }
void m1()
{ System.out.println(super.a
); super.eat();
    }
}
class MethodOverridingDemo {
    public static void main(String[] args)
{ Animal a1 = new Animal();
System.out.println(a1.a);
```

```
a1.eat();
  a1.sleep();
        //method overriding
Dog d1= new Dog();
System.out.println(d1.a);
d1.eat();
d1.sleep();
d1.bark();
        d1.m1();
// Runtime polymorphism/upcasting
Animal obj = new Dog();
obj.eat();
   }
}
```

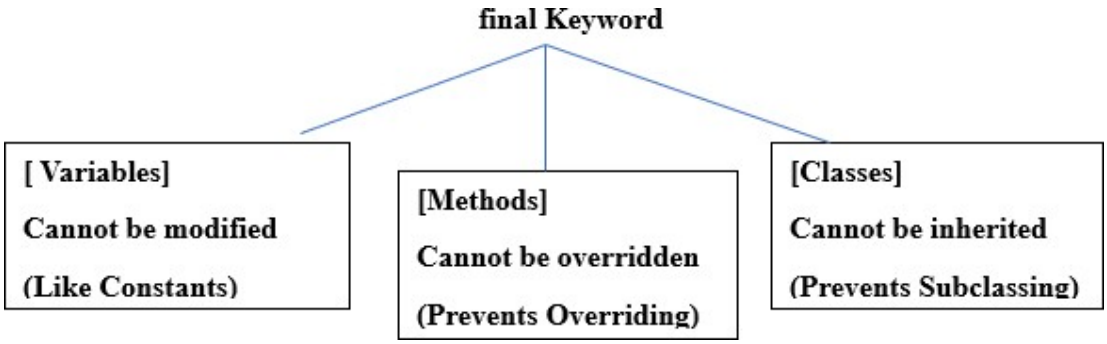
Output:

```
c:\nrcm>javac MethodOverridingDemo.java

c:\nrcm>java MethodOverridingDemo
10
Animal's eat() method
Animal's sleep() method
20
Dog's eat() method
Animal's sleep() method
Dog's bark() method
10
Animal's eat() method
Dog's eat() method
```

final keyword:

There are **three ways** to use the final keyword in Java:

1. **final variables:**cannot be modified. They are treated like constants.
2. **final methods:**cannot be overridden by subclasses,which will prevent overriding.
3. **final classes:**cannot be subclassed,which willprevent inheritance.



1. final variables:

A variable which is declared with final keyword is called final variable.
Final variables can be declared at different scopes. Final variables can be instance variables or static variables or local variables.

Initialization Rules

A final variable does not have to be initialized immediately at declaration, but it **must be assigned a value exactly once** before it is read. If it is left uninitialized at declaration, it is called a **blank final variable**.

- **Local final variables:** Must be initialized before they are used in the method.
- **Instance final variables:** Must be initialized either at declaration, in an instance initializer block, or inside **every** constructor.
- **Static final variables:** Must be initialized either at declaration or inside a static initializer block.

Program 1: Java Program on final variables

```
class Test {
        final int a = 10; //a is instance final variable
final static int b = 20;  //bis static final variable
        public static void main(String[] args)
        {  final int c = 30;  //c is a local final
        variable Test t1=new Test();
        System.out.println(t1.a + "" + t1.b + "" + c);
                //  t1.a=  t1.a + 10; compiler error
                //  t1.b=  t1.b + 20; compiler error
                //  t1.c=  t1.c + 30; compiler error
        }
```

```
}
```

Output:

```
C:\nrcm>javac Test.java

C:\nrcm>java Test
10 20 30
```

2. final methods:

A final method cannot be overridden by subclasses. Although it cannot be overridden, a final method is still inherited by subclasses. You can create another method with the same name but different parameters(overloading). Declaring a method as final ensures its implementation remains unchanged, which is useful for critical or security-sensitive logic.

A method can be both static and final. Since static methods are not overridden, final simply prevents any attempt to redefine the method in subclasses.

A private method can also be declared final. Since private methods are not visible to subclasses, overriding is already impossible, so final is generally redundant but allowed.

A method cannot be both abstract and final, abstract requires subclasses to implement the method, while final forbids overriding, so the two modifiers conflict

Example

```
class Animal {
        final void eat()
      { System.out.println("Animal's eat() method");
        }
}
class Dog extends Animal {
        // Can't override final method
        // Compiler Error
/* void eat() {
System.out.println("Dog's eat() method");
        } */
}
```

Program2: Java Program on final methods

```
// Overloading final method
class Test{
    final void display()
{ System.out.println("No
parameters");
    }
    void display(int x)
{ System.out.println(x);
}
 public static void main(String[] args)
            { Test t1=new Test();
            t1.display();
             t1.display(10);
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
No parameters
10
```

3. final classes:

A final class cannot be extended (subclassed). Declaring a class as final ensures that no other class can modify or override its behaviour through inheritance i.e. prevents inheritance. Prevents malicious or unintended extension of sensitive classes, making the code more secure i.e. improves security.

Many immutable classes are declared final to prevent subclasses from changing their behaviour i.e. maintains immutability.

Example: String is a final class.

A final class can have constructors, instance variables, static variables, methods, and even final methods. A final class can implement one or more interfaces, but it cannot be extended by another class.

Example

```
//final class
final class Box{
        int width, height, depth;
}
  // Compiler Error
class Boxweight extends Box{
……….
}
```

```
// final class
final class Calculator
{ public int add(int a, int b)
{ return a + b;
}
}
class Test {
 public static void main(String[] args)
{ Calculator c = new Calculator();
System.out.println(c.add(10, 20));
}
}
```

Output:

```
c:\nrcm>javac Test.java

c:\nrcm>java Test
30
```

Abstract classes

Classes in Java

Concrete classes	Abstract classes
➢ Must contains only concrete methods ➢ The class should not be declared with abstract keyword	➢ May contain only concrete methods or only abstract or both concrete and abstract methods. ➢ The class should be declared with abstract keyword

Ex:
// only concrete methods
class A{
void m1(){
}
	Ex 1:
	//only concrete methods
	abstract class A{
	void m1(){
	}

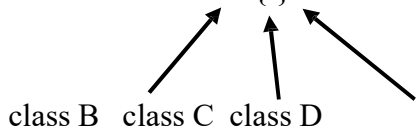
void m2(){
}
}
	}
	
	Ex 2:
	//only abstract methods
	abstract class
	B{ abstract void m1();
	abstract void m2();
	}
	
	Ex3:
	//concrete and abstract methods
	abstract class A{
	void m1(){
	}
	abstract void m2();
	}

Rules for abstract classes:

1. Abstract classes may contain only concrete methods or only abstract methods or both.
2. Abstract classes may contain concrete methods but a class which contains atleast one abstract method must be declared as abstract otherwise compiler error.

   Ex:
   class C {
   abstract void m1();
   }
   Output: Compiler Error
3. Abstract classes must be extended.
4. There must be atleast one subclass which extends the abstract class

Ex: abstract class A { }



5. A class which extends the abstract class must provide implementation to all the abstract methods of the abstract class.
6. If the subclass doesn't want to implement all the abstract methods of the abstract class then declare the subclass as abstract and this subclass (abstract) must be extended by atleast one class.

7. For abstract classes we cannot create objects with new operator, we can only create reference variables.

Ex:

```
abstract class A {
}
A a1; // Reference variable
A a1 = new A(); // compiler error
```

8. **Upcasting:** Upcasting is required to access the members of the abstract class because for abstract classes we cannotcreate objects with new operator.
9. Abstract methods provide more sharability when compare to concrete methods.

A m1();

B m1(){} C m1(){}D m1(){}

One structure can be used by many classes

Program 1: Java Program using abstract classes

```
abstract class A
{ void m1() {
System.out.println("A's m1()");
}
abstract void m2();
abstract void m3();
}
class B extends A
{ void m2(){
System.out.println("B's m2()");
 }
void
m3(){ System.out.println("B's
m3()");
}
void
m4(){ System.out.println("B's
m4()");
 }
}
class AbstaractDemo {
public static void main (String args[]) {
A a1 = new B(); // Upcasting
 a1.m1();
```

```
// a1.m4(); Compiler error
B b1 = new B(); // subclass obj
b1.m1();
b1.m2();
b1.m3();
b1.m4();
}
}
```

Output:

```
C:\nrcm>javac AbstractDemo.java

C:\nrcm>java AbstractDemo
A's m1()
B's m2()
B's m3()
A's m1()
B's m2()
B's m3()
B's m4()
```

Program 2: Java Program using abstract classes

```
abstract class A
  { void m1() {
System.out.println("A's m1()");
 }
abstract void m2();
abstract void m3();
}
abstract class B extends A
{ void
m2(){ System.out.println("B's
m2()");
}
          // abstract void m3();
void
m4(){ System.out.println("B's
m4()");
 }
}
class C extends B
{ void m3(){
System.out.println("C's m3()");
 }
```

```
}
}
class AbstractClassDemo {
public static void main (String
args[]){ A a1 = new C(); // Upcasting
a1.m1();a1.m2();a1.m3();
                    B b1 = new C();// Upcasting
b1.m1();b1.m2();b1.m3();b1.m4();
C c1 = new C();
c1.m1();c1.m2(); c1.m3(); c1.m4();  c1.m5();
        }
}
```

Output:

```
C:\nrcm>javac AbstractClassDemo.java

C:\nrcm>java AbstractClassDemo
A's m1()
B's m2()
C's m3()
A's m1()
B's m2()
C's m3()
B's m4()
A's m1()
B's m2()
C's m3()
B's m4()
c's m5()
```