JAVA

What is java…?

- Java is a high level object oriented programming language .
- Java was released by sun microsystems in 1995.
- The current version of java is 17 which was released on september 14th 2021.
- Java is very versatile as it is used for developing applications on the web, mobile, desktop, etc. using different platforms. Also, java has many features such as dynamic coding, multiple security features, platform-independent characteristics, etc.

SYLLABUS

- ✓ What is Programming Language.
- ✓ What is JDK,JVM,JRE.
- ✓ TOKEN and types of TOKEN.
- ✓ What is a Variable
- ✓ What is a DataType
- ✓what are Operators
- ✓what is a Method?

- ✓ What is Dynamic reading?
- ✓ What is Control Statements?
- ✓ What are loops?
- ✓ What are Static context?
- ✓ What are non static context?

This will be the end of your PART-1………!!

Concepts in oops

- ✓What is Encapsulation.
- ✓What is Relationships(inheritance).
- ✓ What is Polymorphism.
- ✓ What is Abstraction.
- ✓ What is a Interface.

This is the end of oops (Part-2).......!!!

Part-3

- ✓ What is a Object class.
- ✓ What is a String class.
- ✓ What are Arrays.
- ✓ What is Exception handling.
- ✓ What are Collections.
- ✓ What is Multithreading.

WHAT IS PROGRAMMING LANGUAGE?

- A language or a medium which is used to instruct a computer to perform some particular task is known as Programming language.

- Example: Java, html, C, etc.

TYPES OF PROGRAMMING LANGUAGE:

➨1. High Level languages

➨2. Mid Level languages

➨3. Low Level languages

LOW LEVEL LANGUAGE:

- The language which is easily understandable by the processor is known as Machine level language or low level language
- Binary language is the example of Low level language, i.e.. 0’s and 1’s.

ASSEMBLY LEVEL LANGUAGE

- The Machine level language consists of some predefined words known as mnemonics.
- It is an intermediate level language as it is understandable to some extent by the processor as well by humans.
- Assembler is used to convert assembly level language into machine understandable language.

HIGH LEVEL LANGUAGE

- A language that is easy, readable, and understandable by human is known as High Level Language.

- Ex: java, python, C, C++ etc.

- Easy,
- Readable,
- Understandable

JDK(JAVA DEVELOPMENT KIT)

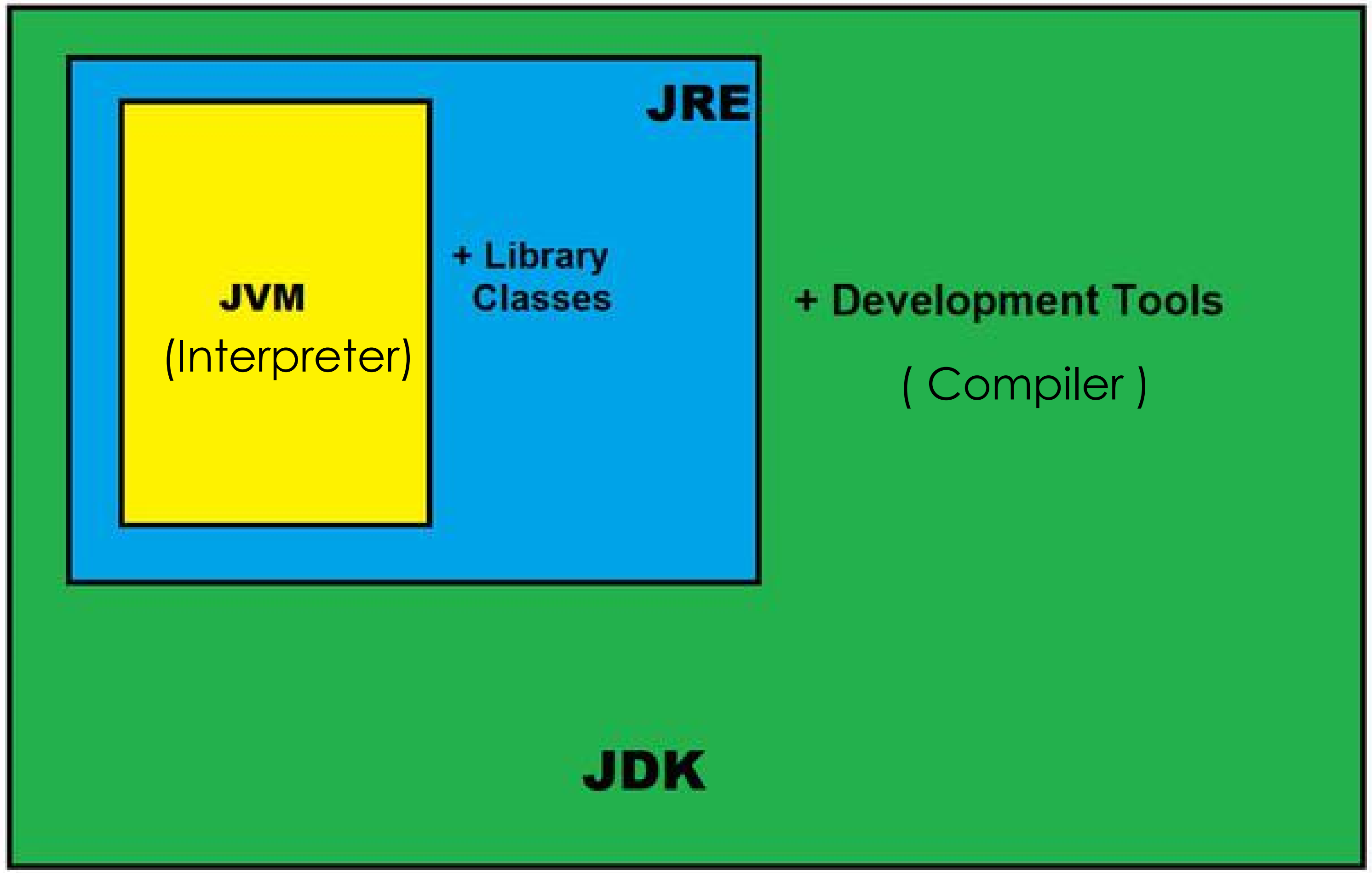
- JDK-is a package which consist of java development tools like java compiler and JRE .

WHAT IS JRE ?

- JRE is an environment which consist of JVM and built in class which is required for the execution of java program.

WHAT IS JVM?

- JVM is responsible for actual execution of a java program.
- It converts the class files into low level language with help of interpreter.


JRE
JVM
(Interpreter)
+ Library Classes
+ Development Tools
(Compiler)
JDK

PLATFORM DEPENDENT

AND

PLATFORM INDEPENDENT

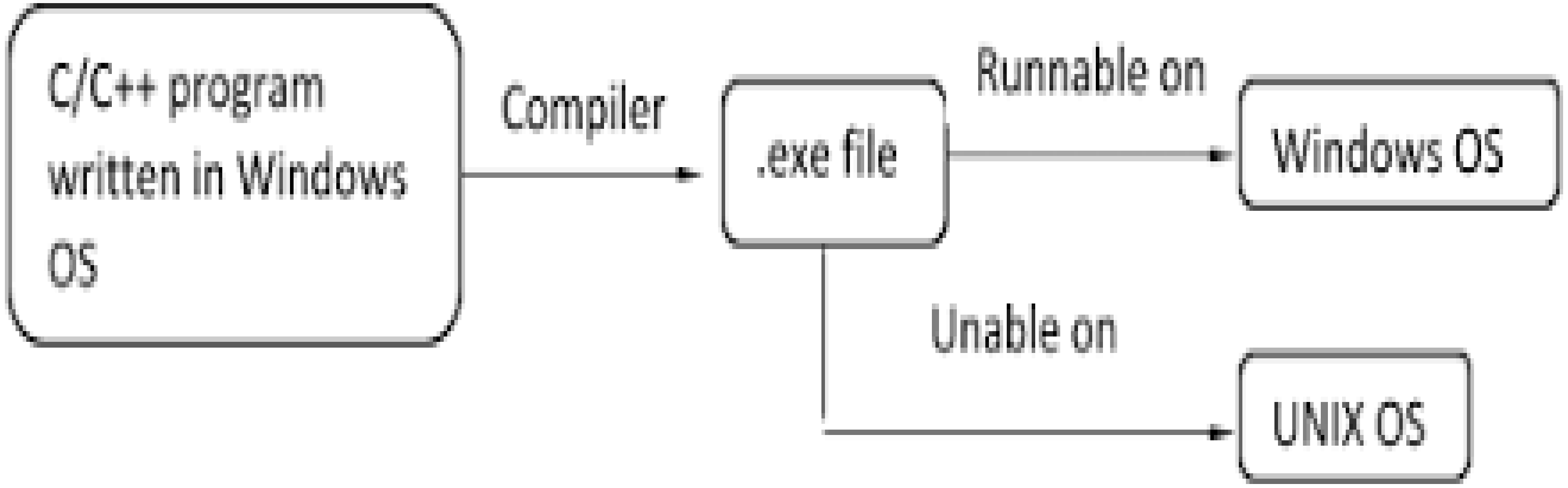
PLATFROM DEPENDENT

- Platform dependent typically refers to applications that run under only one operating system in one series of computers (one operating environment).

- If we develop one application using Windows Operating System, then that application can only be executed on Windows Operating System and cannot be run on other Operating Systems like Mac, Linux etc. This is called platform dependency.

- And the languages used for developing such applications is called Platform Dependent language.

- C and C++ are platform-dependent languages.

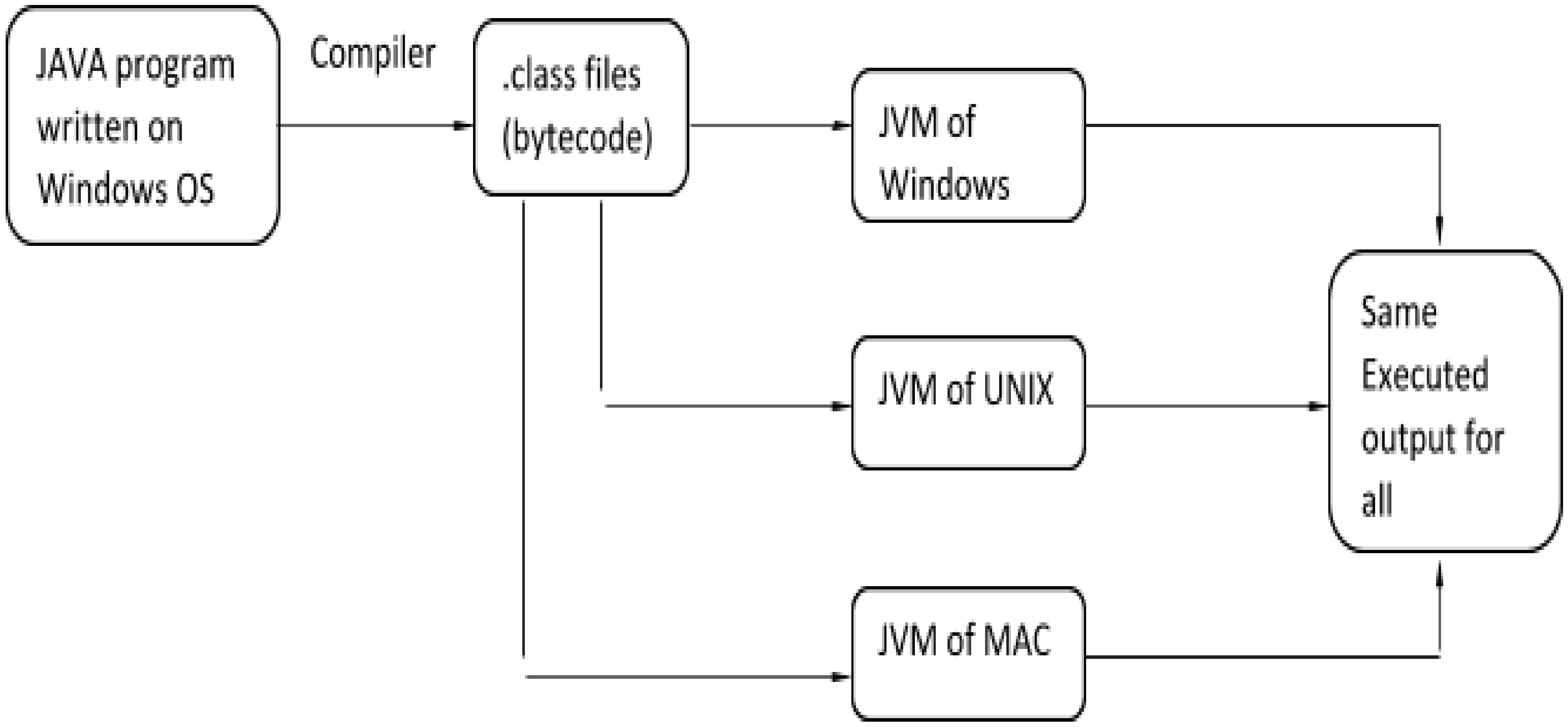

C/C++ program written in Windows OS
Compiler
.exe file
Runnable on
Windows OS
Unable on
UNIX OS

- Platform independent typically refers to applications that works on any platform (operating system) without needing any modification.

- When you write a program in JAVA and compile it, a class file is generated, which consists of byte code. This class file will not be in executable stage.

- The main purpose of generating byte code for a compiled program is to achieve platform independency .

- It means, this byte code generated in one platform(for example, windows OS) can be executed in other platforms such as MAC OS, LINUX OS etc.


JAVA program written on Windows OS
Compiler
.class files (bytecode)
JVM of Windows
JVM of UNIX
JVM of MAC
Same Executed output for all

BASIC STRUCTURE OF A JAVA PROGRAM

- Java instructions are always written inside the class. The syntax is :

Class block

```
class  classname
{
        public static void main(String[] args)
        {
                // statements
        }
}
```

Save file as : classname.java

- Every class in java must have a name. known as class name.
- Every class has a block, known as class block.

- Any class in java can be executed only if main method is created, as shown below :

- Syntax to create a main method :\

```
class Example
{
        public static void main(String[] args)
        {
                // statements
        }
}
```

Main method

Note:

- We can create a class in java without a main method.
- It is compile time successful , so class file will be generated.
- But we cannot execute the class file.

```
class  classname
{
        // statements
}
```

To execute a java program,

Step 1: **Save** the program with the same name as class name (Recommended) with .java extension.

Program1.java

Step 2: Compile the java program using the command –

```
javac filename.java
```

Step 3: Execute the java program using the command –

```
java filename
```

Example program 1:

Write a java program to print hello world

```
class  HelloWorld
{
        public static void main(String[] args)
        {
                System.out.println(“Hello World”);
        }
}
```

OUTPUT :

```
Hello World
```

Example program 2:

Write a java program to print your name and phone number.

```
class  MyDetails
{
        public static void main(String[] args)
        {
                System.out.println(“naMe  : XYZ”);
                System.out.println(“NUMBER  : 123456789”);
        }
}
```

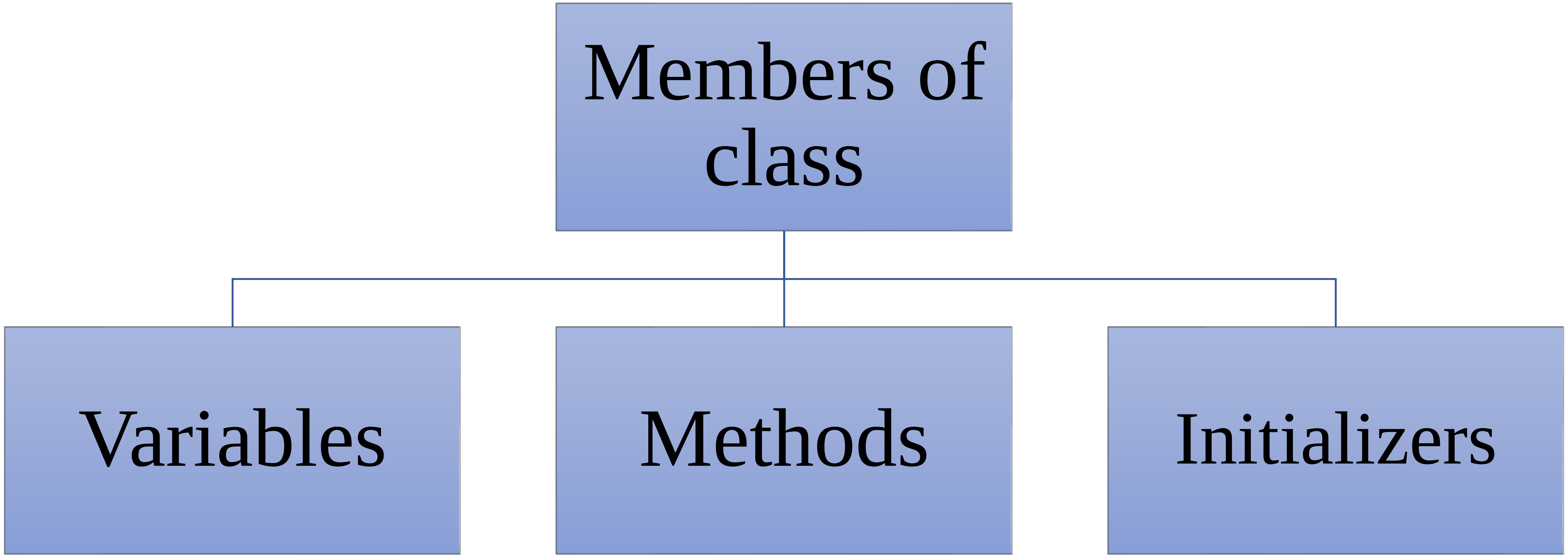
OUTPUT :

```
naMe : XYZ
NUMBER : 123456789
```

Activity 1:

- Write a java program to print hello and good morning.
- Write a java program to print name, percentage and year of pass out.

MEMBERS OF A CLASS



Variables :

- A variable is a container which is used to store data.

Methods :

- A method is a block of instructions which is used to perform a task.

Initializers:

- Initializers are used to execute the start up instructions.

Print statement :

- print statement only prints the data but does not move the cursor to a new line.
- Example : System.out.print(data);
- We cannot use print statement without passing any data.
- System.out.print( ); (Not Allowed)

Println statement :

- println statement prints the data and also moves the cursor to a new line.
- Example : System.out.println(data);
- We can use println statement without passing any data. It will just print new line.
- System.out.println( ); (Allowed)

Program to print Hello and Good Morning using print :

```
class  HelloWorld
{
        public static void main(String[] args)
        {
                System.out.print(“Hello World”);
                System.out.print(“Good Morning”);
        }
}
```

OUTPUT :

```
Hello WorldGood Morning
```

Activity 2:

- Write a java program to print hello and good morning using print statement .
- Write a java program to print your name, percentage and year of pass out using print statement.
- Write a java program having a println statement without passing any data.
- Write a java program having a print statement without passing any data.

TOKENS

In Java, Tokens are the smallest elements of a Java program.

The Java compiler breaks the line of code into text (words) is called **Java tokens**.

For example,

```
class Demo
{
        public static void main(String args[])
        {
                System.out.println(" Hello ");
        }
}
```

In the above code , class, Demo, {, static, void, main, (, String, args, [,],), System, out, ., println, Hello are all the Tokens.

Types of Tokens

Java token includes the following:

- Keywords
- Identifiers
- Literals
- Operators etc.

Keywords

- These are the pre-defined, reserved words which the java compiler can understand.
- Each keyword has a special meaning, i.e.. Each keyword is associated with a specific task.
- A programmer cannot change the meaning of any keyword.
- We have 50+ keywords in java. Few keywords are class, public ,static, void etc.

Note : It is always written in lower case.

Identifier

- It is a name given to the components of java by the programmer.

- Components of java are :
  1. class
  2. variables
  3. methods
  4. interfaces etc.

- Identifiers are defined by the programmer.

Note : A programmer should follow some rules and conventions for defining identifiers.

Rules to define identifiers :

- Identifiers cannot start with number (but may contain numbers).
- It cannot have any special characters except _ and $.
- The whitespace cannot be included in the identifier.
- Identifier name must be different from the keywords (We cannot use keywords as identifiers).

- **Some valid identifiers are** :

1. PhoneNumber
2. PRICE
3. radius
4. a
5. hello123
6. _phonenumber
7. $circumference
8. jagged_array

Conventions

- The coding or industrial standards which are highly recommended to be followed by the programmer is known as conventions.

Note : Compiler will not validate the conventions. Therefore, if conventions aren’t followed, then we will not get compile time error. But it is highly recommended to follow the conventions.

Convention for class name :

- **Single word** : If the class name is a single word, then the first letter must be in upper case and remaining letters in lower case.

  For example, class Addition etc.

- **Multi word** : If the class name is a multi word, then the first letter of each word must be in upper case and remaining letters in lower case.

  For example, class AdditionOfTwoNumbers,

  class SquareRoot, etc.

ACTIVITY 3

1. Write a java program with the class name starting from a number.
2. Write a java program with the class name same as a keyword.
3. Write a java program with the class name starting with $.
4. Write a java program with the class name starting with _.
5. Write a java program with the class name starting with any special character except _ and $.
6. Write a java program with class name as a multi word.

Literals :

- The values or data used in a java program is known as literals.

- The data is generally categorized into two types :
  1. Primitive values
  2. Non Primitive values

- **Primitive values** : Single value data is called as primitive values.

→ Following are the primitive values :

a. **Number literals** (integer number literals (1,4,18 etc), floating number literals (1.0, 19.5, 45.9 etc).

b. **Character literals** (Anything enclosed within single quotes).

c. **Boolean literals** (false, true).

- **Non-Primitive values** : Multi value data (group of data) is known as non primitive value.

- Example of non primitive values are String, Object reference.

- **String literals** : Anything enclosed within double quotes ,i.e.. (“ “) is known as string literals.

→ The length of a string can be anything.

→ Strings are case sensitive.

→Example : “hello”, “TRUE”, “123”, “HelLo”, “Oops@” etc.

Scope of a variable

- Scope of a variable is the part of the program where the variable is accessible.

(or)

- The visibility of a variable is known as Scope of a variable.
- Based on scope of variables, we can categorize variables into three types :

1. Local variables
2. Static variables
3. Non Static variables

Local Variable :

- The variable declared inside a method block or any other block except class block is known as a local variable.

- **Characteristics of a local variable :**

→ We cannot use local variables without initialization. If we try to use local variables without initialization, then we will get compile time error.

→ Local variables will not be initialized with default values.

→ The scope of variables is nested inside the block wherever it is declared. Hence, it cannot be used outside the block.

- **Example :**

```
class Example
{
        public static void main(String[] args)
        {
                int i=10;  // integer type variable i
                float f=10.5;  // float type variable f
                char ch=‘a’;  // character type variable ch
                boolean b=true;  // boolean type variable b
                System.out.println(i);
                System.out.println(f);
                System.out.println(ch);
                System.out.println(b);
        }
}
```

PRIMITIVE DATA TYPES :

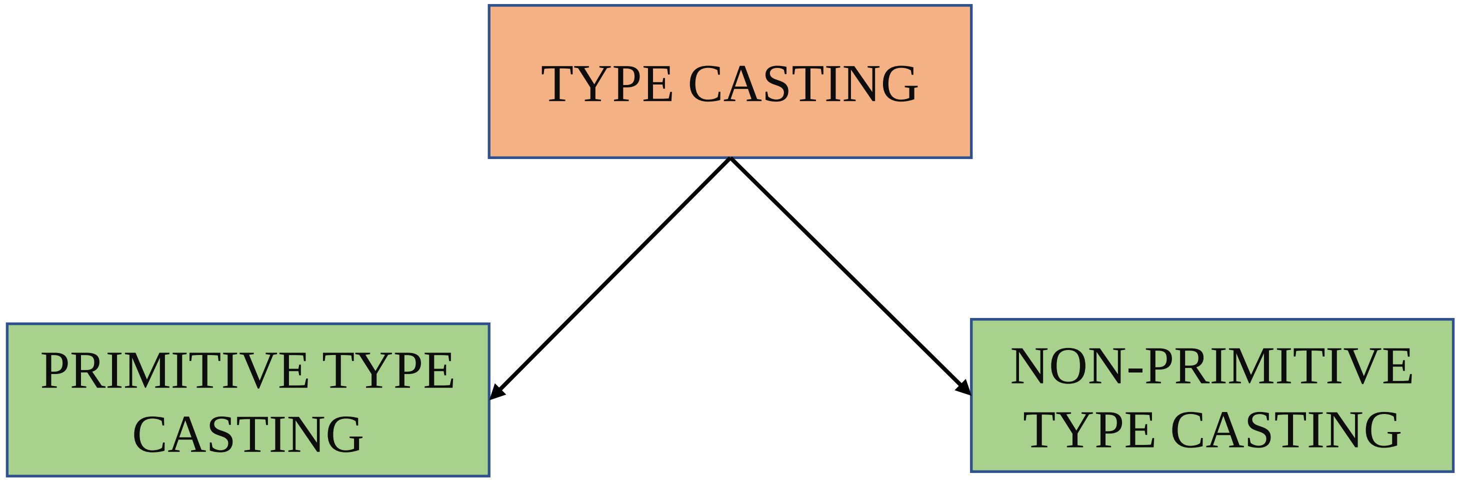
byte	1 byte (stores numbers from -128 to 127)
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
boolean	1 bit (true or false) (0 or 1)
char	2 bytes

Byte → short → char → int → long → float → double

TYPE CASTING

What is type casting ?

- The process of converting one type of data is known as type casting.
- There are two types of type casting.



Primitive Type Casting

- The process of converting one primitive type value into another primitive type value is known as primitive type casting.

- There are two types of primitive type casting.

1. **WIDENING**
2. **NARROWING**

WIDENING :

- The process of converting smaller range primitive data type into higher range primitive data type is known as widening.
- In this process, there is no data loss.
- Since there is no data loss, compiler implicitly performs widening. Hence, it is also called as auto-widening.

Byte → short → char → int → long → float → double

Example for widening :

```
class Widening
{
        public static void main(String[] args)
        {
                int a=10;
                float num=a;  // automatically int is converted to float type
                System.out.println(a);
                System.out.println(num);
        }
}
```

Output :

```
10
10.0
```

```
class Widening
{
        public static void main(String[] args)
        {
                int a=10;
                char c=a;  // Compile time error
                System.out.println(a);
                System.out.println(c);
        }
}
```

Note : It will give compile time error since we are trying to convert higher range data type (int) to smaller range data type(char).

NARROWING :

- The process of converting higher range primitive data type into lower range primitive data type is known as widening.
- In this process, there is possibility of data loss.
- Since there is possibility for data loss, compiler will not perform anything implicitly.
- The programmer will have to explicitly(manually) perform Narrowing.
- This can be achieved with the help of type cast operator.

Byte ← short ← char ← int ← long ← float ← double

Example :

```
class Narrowing
{
        public static void main(String[] args)
        {
                int a=10;
                char ch=a;  // Compile time error
                System.out.println(a);
                System.out.println(c);
        }
}
```

Note : It will give compile time error since we are trying to convert higher range data type (int) to smaller range data type(char) without type cast operator.

Type cast operator :

- It is an unary operator.
- It is used to explicitly convert one data type to another data type.

Example for narrowing :

```
class Narrowing
{
        public static void main(String[] args)
        {
                int a=10;
                char ch=( char ) a; //explicitly converting float to int using type cast operator
                System.out.println(a);
                System.out.println(c);
        }
}
```

Output :

```
10.0
10
```

OPERATORS :

Operators are predefined symbols which are used to perform some specific task on the given data.

The data given as input to the operator is known as operand.

Based on the number of operands, there are three types of operators :

→ Unary operator

→ Binary operator

→ Ternary operator

- Based on the task performed, there are 8 types of operators :

  → Arithmetic operators
  → Relational operators
  → Logical operators
  → Assignment operators
  → Conditional operator
  → Bitwise operators
  → Miscellaneous operators
  → Increment/Decrement operators

Ternary operator

- Ternary operator is also known as the conditional operator.
- It is used to handle simple situations in a line.
- It accepts three operands .
- Syntax :

  condition ? statement1 : statement2 ;

- The above syntax means that if the value given in condition is true, then statement2 will be executed; otherwise, statement3 will be executed.
- The return type of condition is boolean.
- The return type of conditional operator depends on statement1 and statement2.

Example :

```
int val=5;
String result   =   val==5 ? “you are right” : “you are wrong”;
```

OUTPUT:

```
You are right
```

```
class ConditionalOperator
{
        public static void main(String[] args)
        {
                int a = 6, b = 12;
                int large = a>b ? a :  b;
                System.out.println(" Largest number is : "+large);
        }
}
```

OUTPUT:

```
Largest number is 12
```

INCREMENT / DECREMENT OPERATOR

Increment operator is used to incrementing a value by 1. There are two varieties of increment operator:

- **Post-Increment:** Value is first used for computing the result and then incremented. Example (a++)
- **Pre-Increment:** Value is incremented first and then the result is computed. Example (++a)

Decrement operator is used for decrementing the value by 1. There are two varieties of decrement operators.

- **Post-decrement:** Value is first used for computing the result and then decremented. Example (a--)
- **Pre-decrement:** Value is decremented first and then the result is computed. Example (--a)

PRE INCREMENT / DECREMENT

- We can achieve pre increment or decrement with the help of three steps :

  → Increment / decrement by 1

  → Update the value

  → Substitute the value

Example for pre increment :

```
class  PreInc
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = ++a;
        System.out.println(b);
        System.out.println(++b);

    }
}
```

Output :

```
11
12
```

Example for pre decrement :

```
class  PreInc
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = --a;
        System.out.println(b);
        System.out.println(--b);

    }
}
```

Output :

```
9
8
```

POST INCREMENT / DECREMENT

- We can achieve post increment or decrement with the help of three steps :

  → substitute the value

  → increment / decrement by 1

  → update the value.

Example for post increment :

```
class  PreInc
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = a++;
        System.out.println(b);
        System.out.println(b++);
        System.out.println(b);
    }
}
```

Output :

```
11
11
12
```

Example for post decrement :

```
class  PreInc
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = a--;
        System.out.println(b);
        System.out.println(b--);
        System.out.println(b);
    }
}
```

Output :

```
9
9
8
```

METHODS

- **Method** is a block of instructions which is used to perform some specific task.

- **Syntax to define any method is :**

```
[access modifier] [modifier] returntype methodname ([formal arguments])
{


}
```

It consists of three parts:

1. Method signature
2. Method declaration
3. Method definition

1. Method signature :

→ Method name

→ Formal arguments

2. Method declaration :

→ Access modifier

→ Modifier

→ Return type

→ Method signature

3. Method definition :

→ Method declaration

→ Method body / Method block

MODIFIERS :

- Modifiers are keywords which are responsible to modify the characteristics of the members.

- Example : static, abstract, final, synchronized, volatile, transient etc.

- **static :**

  →The users can apply static keywords with variables, methods, blocks, and nested classes.

  →The static keyword belongs to the class .

  →The static keyword is used for a variable or a method that is the same for every instance of a class.

ACCESS MODIFIER

- Access modifiers are keywords used to change the accessibility of members of class.
- There are four levels of access modifiers :

  → private

  → public

  → protected

  → default

public :

- The public access modifier is specified using the **keyword public**.
- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program.

- Any method , after execution can return a value back to the caller.
- Therefore, if a method is returning any value, it is mandatory to specify what type of value is returned by the method.
- It must be specified in the method declaration statement.
- This is done with the help of return type.

RETURN TYPE :

- Return type is a data type which specifies what type of data is returned by the method after execution.
- A method can have the following data types :

  → Primitive data types

  → Non-Primitive data types

  → void

void : It is a data type which is used as a return type when the method is returning nothing.

Note : A method can have any number of methods.

Note : A method cannot be created inside another method.

Example :

```
class Example
{
        public static void add()
        {
                int a=10;
                int b=20;
                int sum=a+b;
                System.out.println("sum is : "+sum);
        }
}
```

RETURN STATEMENT :

- A method can return a data back to the caller with the help of return statement.
- **return :** It is a keyword.
- It is a control transfer statement.
- When the return statement is executed, the execution of the method is terminated ant control is transferred back to the calling method.

Steps to use return statement :

→Provide a return type for a method.

→ Use the return statement in the value to be returned.

Note : The type specified as return type must be same as the type of value passed in a return statement.

- A method will get executed only when it is called. We can call a method with the help of method call statement.

METHOD CALL STATEMENT :

- The statement which is used to call a method is known as method call statement.
- Syntax to create a method call statement is :

```
methodName ( [ Actual arguments ]);
```

FLOW OF METHOD CALL STATEMENT :

1. Execution of calling method is paused.
2. Control is transferred to the called method.
3. Execution of called method begins.
4. Once the execution of called method is completed, the control is transferred back to the calling method.
5. Execution of calling method resumes.

Example :

```
class Example
{
        public static void m1()
        {
                System.out.println("From m1 method" );
        }
        public static void main(String[] args)
        {
                System.out.println("start" );
                m1();
                System.out.println("end" );
        }
}
```

Output :

```
start
From m1 method
end
```

TYPES OF METHODS :

- There are two types of method.
  - → Parameterized method
  - → No argument method

1. **Parameterized method :** The method which consists of formal arguments is known as parameterized method.

   The parameterized methods are used to accept the data.

2. **No argument method :** The method which doesn't consists of any arguments is known as no argument method.

Formal Arguments : Variables which are declared in method declaration statement are known as formal arguments.

Actual Arguments : The values passed in the method call statement are known as actual arguments.

RULES TO CALL THE PARAMETERIZED METHOD

- The number of actual arguments should be same as the number of formal arguments.

- The type of corresponding actual argument should be same as the type of formal argument.

- If not, the compiler tries to implicitly perform conversion. If it is not possible, then we will get compile time error.

DECISION MAKING STATEMENTS (Control statements)

- Decision making statements help the programmer to skip a block of instruction from execution if the condition is true.

- Types of decision making statements :

    → if statement

    → if else statement

    → if else-if statement

    → switch statement

If statement :

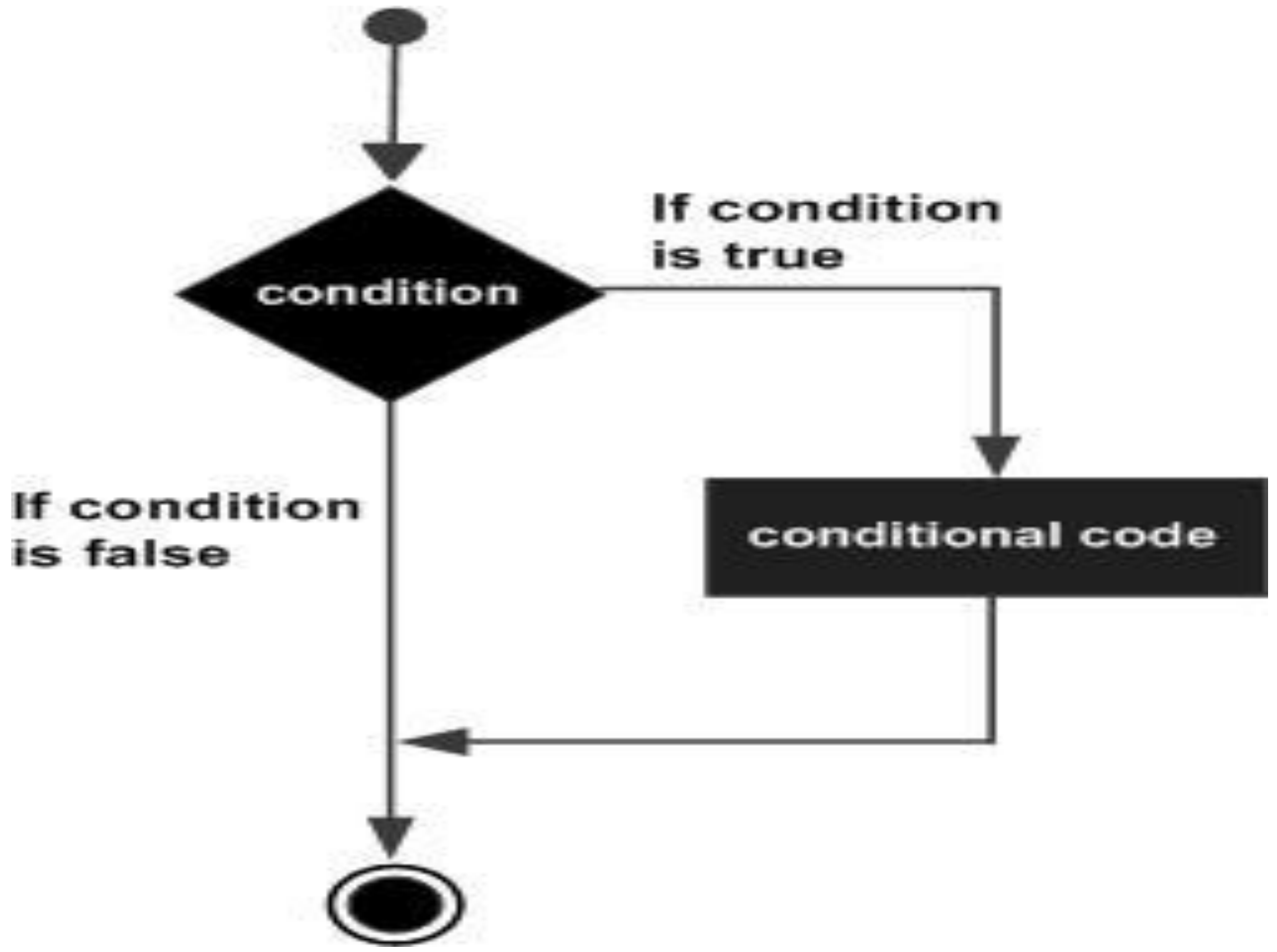
- An if statement consists of a Boolean expression followed by one or more statements.

- **Syntax :**

```
if(condition)
{
        // Statements will execute if the Boolean expression is true
}
```

- **Work flow :**

  → If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed.



- **Example :**

Output :

This is if statement

```
class Test
    {
        public static void main(String [] args)
        {
            int x = 10;
            if( x < 20 )
            {
                System.out.print("This is if statement");
            }
        }
    }
```

If else statement :

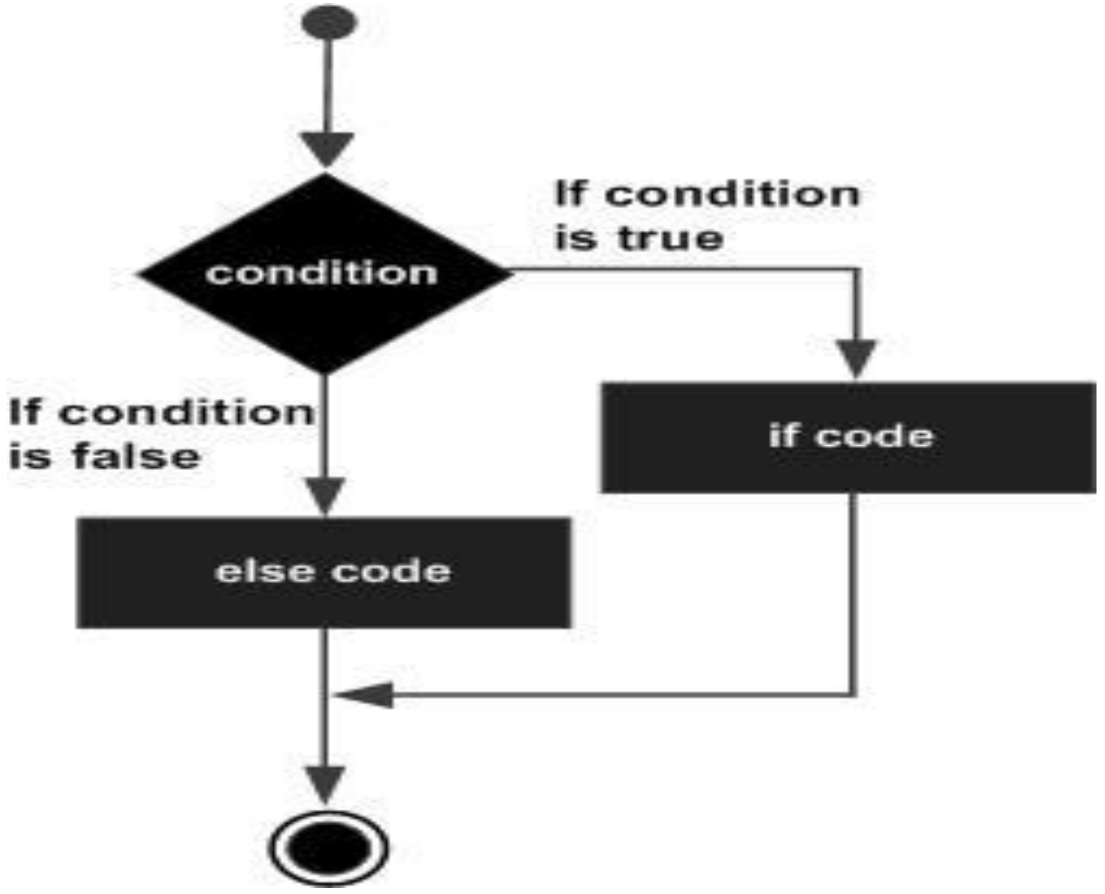
- An **if** else statement consists of a if , followed by an **else** statement.
- **Syntax :**

```
if(condition)
{
        // Statements will execute if the Boolean expression is true
}
else
{
        // Statements will execute if the Boolean expression is false
}
```

- **Work flow :**

→ if the condition is true, then the instructions inside if block will get executed.

→ if the condition is false, then the instructions inside else block will get executed.



- **Example :**

Output :

This is else statement

```
class Test
      {
            public static void main(String args[])
            {
                  int x = 30;
                  if( x < 20 )
                  {
                        System.out.print("This is if statement");
                  }
                  else
                  {
                        System.out.print("This is else statement");
                  }
            }
      }
```

If else – if statement :

- It consists of an if statement followed by *else if...else* statement, which is very useful to test various conditions.

- **Syntax :**

```
if(condition1)
{
     // Executes when the Boolean expression 1 is true
}
else if(condition2)
{
     // Executes when the Boolean expression 2 is true
}
else if(condition3)
{
\    // Executes when the Boolean expression 3 is true
}
else
{
     // Executes when the none of the above condition is true.
}
```

- **Example :**

```
class Test
         {
                  public static void main(String args[])
                  {
                           int x = 30;
                           if( x == 10 )
                           {
                           System.out.print("Value of X is 10");
                           }
                           else if( x == 20 )
                           {
                           System.out.print("Value of X is 20");
                           }
                           else if( x == 30 )
                           { System.out.print("Value of X is 30");
                           }
                           else
                           { System.out.print("This is else statement");
                           }
                  }
}
```

Switch statement :

- A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.
- **Syntax :**

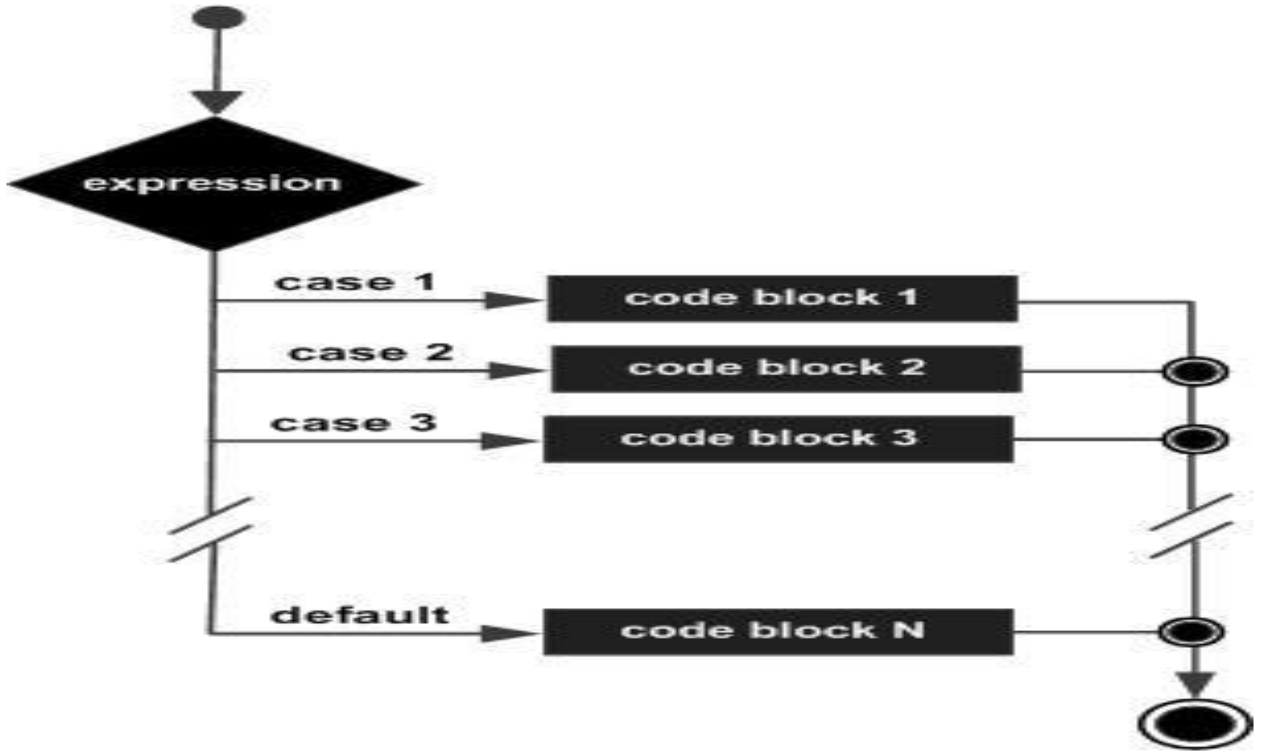
```
switch(expression)
{
        case value :
                // Statements;
        // optional case value :
                // Statements;
        .
        .
        .
                // You can have any number of case statements.
        default :
                // Statements
}
```

- **Work flow :**

→ the value /var /expression passed in the switch gets compared with the value passed in the case from top to bottom order.

→ if any of the case is satisfied, the case block will get executed and all the blocks present below gets executed.

→ if no case is satisfied, the default block gets executed.



The following rules apply to a switch statement −

- You can have any number of case statements within a switch.
- The value for a case must be the same data type as the variable in the switch .
- for a switch, we cannot pass long, float, double, boolean.
- For a case, we cannot pass variables.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- **Break statement :**

  → it is a keyword.

  → it is a control transfer statement.

  → when the break statement is executed, control is transferred outside the block.

Syntax :

```
switch(expression)
        {
                case value :
                        // Statements;
                break;
                // optional case value :
                        // Statements;
                break;
                .
                .
                . // You can have any number of case statements.

                default :
                        // Statements
        }
```

- **Example :**

```
class Test
          {
                    public static void main(String args[])
                    {
                              char grade = 'C';
                              switch(grade)
                              {
                                        case 'A' :
                                                            System.out.println("Excellent!");
                                        break;
                                        case 'B' :
                                                            System.out.println("Well done");
                                        break;
                                        case 'C' :
                                                            System.out.println("Good");
                                        break;
                                        case 'D' :
                                                            System.out.println("You passed");
                                        break;
                                        case 'F' :
                                                            System.out.println("Better try again");
                                        break;
                                        default : System.out.println("Invalid grade");
                              }
                    }
}
```

LOOPING STATEMENTS

- Looping is a feature which facilitates the execution of a set of instructions repeatedly while some condition evaluates to true.

- Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

→While loop
→Do while loop
→For loop

While loop :

- **While loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax:

```
while (test_expression)
{
        // statements


        update_expression;
}
```

The various **parts of the While loop** are:

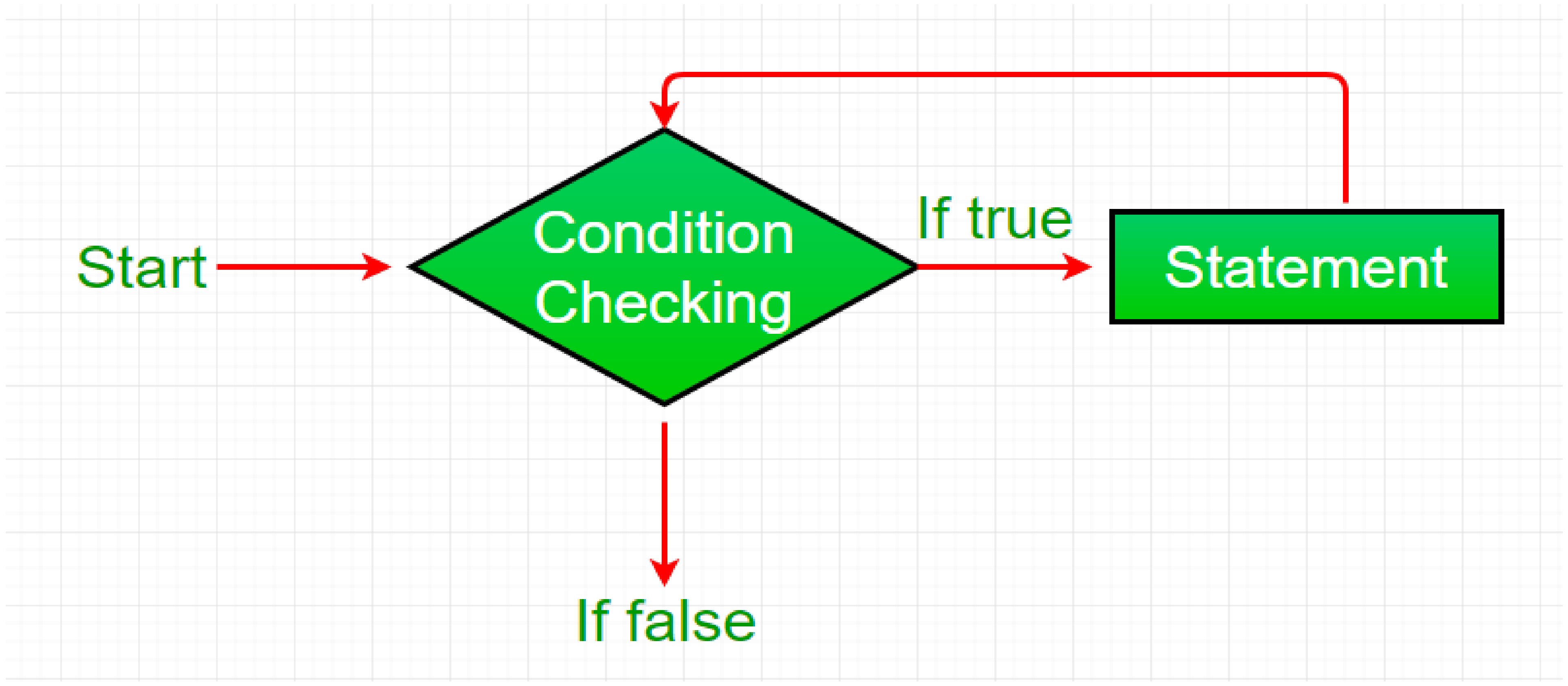
1.Test Expression: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression.
Otherwise, we will exit from the while loop.

Example: i <= 10

2.Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example: i++;

Flow chart while loop (for Control Flow):



- **Example : this program will print numbers from 1 to 10**

```
class whileLoopDemo
{
    public static void main(String args[])
    {
        // initialization expression
        int i = 1;

        // test expression
        while (i < =10)
            {
            System.out.println(i);

            // update expression
            i++;
        }
    }
}
```

Do while loop :

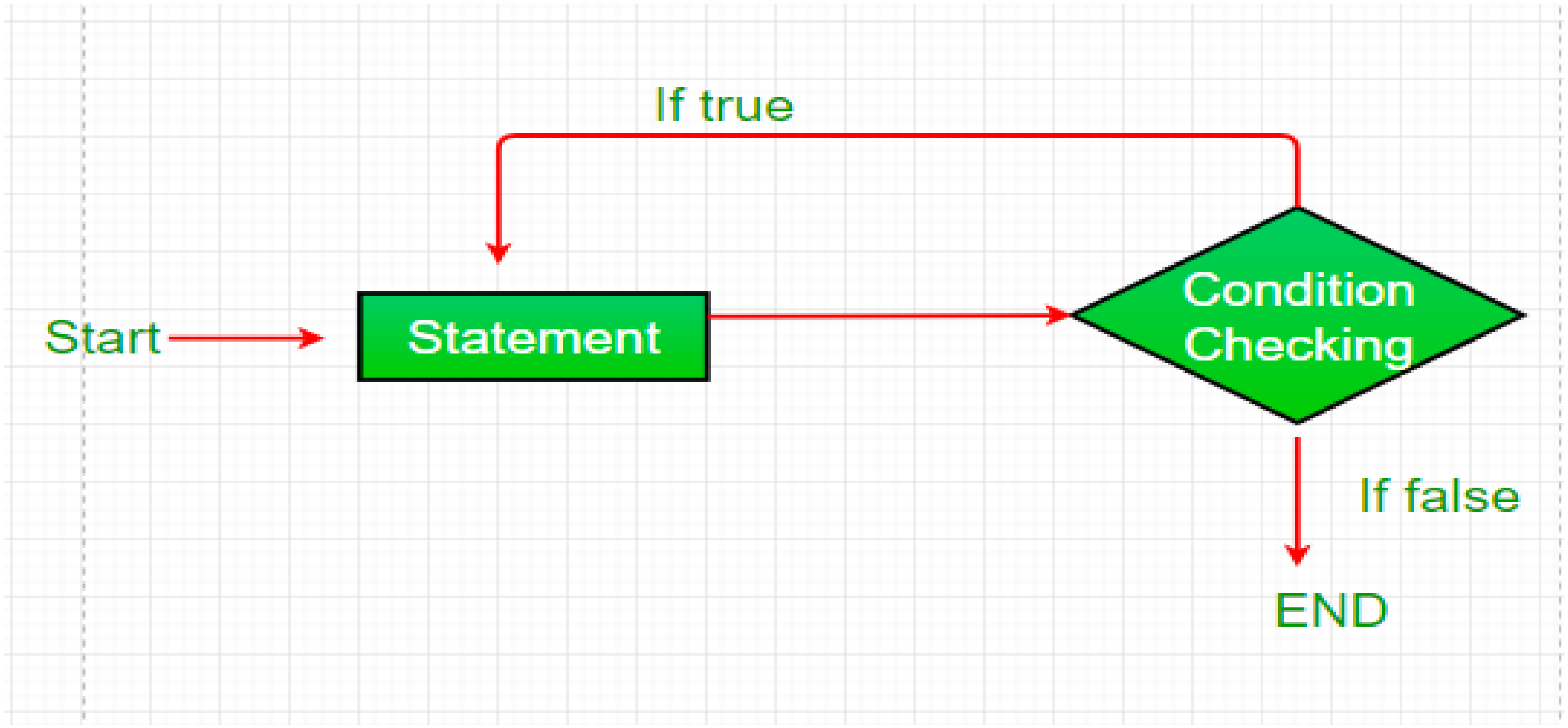
- **do-while loop** is an **Exit control loop**. Therefore, unlike for or while loop, a do-while check for the condition after executing the statements or the loop body.

Syntax:

```
do
        {
                // statements

                update_expression ;
        }
        while (test_expression);
```

Flowchart do-while loop:



- **Example : this program will print numbers from 1 to 10**

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

        // initialisation expression
        int i = 1;
        do {

            // Print the statement
            System.out.println(i);

            // update expression
            i++;
        }
        // test expression
        while (i <=10);
    }
}
```

Difference between while and do while loop:

while	do-while
• Condition is checked first then statement(s) is executed.	• Statement(s) is executed at least once, thereafter condition is checked.
• It might occur statement(s) is executed zero times, If condition is false.	• At least once the statement(s) is executed.
• If there is a single statement, brackets are not required.	• Brackets are always required.
• while loop is entry controlled loop.	• do-while loop is exit controlled loop.

For loop :

- **for loop** provides a concise way of writing the loop structure. The for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

Syntax:

```
for (initialization expr; test expr; update exp)
{
        // body of the loop
        // statements we want to execute
}
```

The various parts of the For loop are:

1. **Initialization Expression:** In this expression, we have to initialize the loop counter to some value.

Example: `int i=1;`

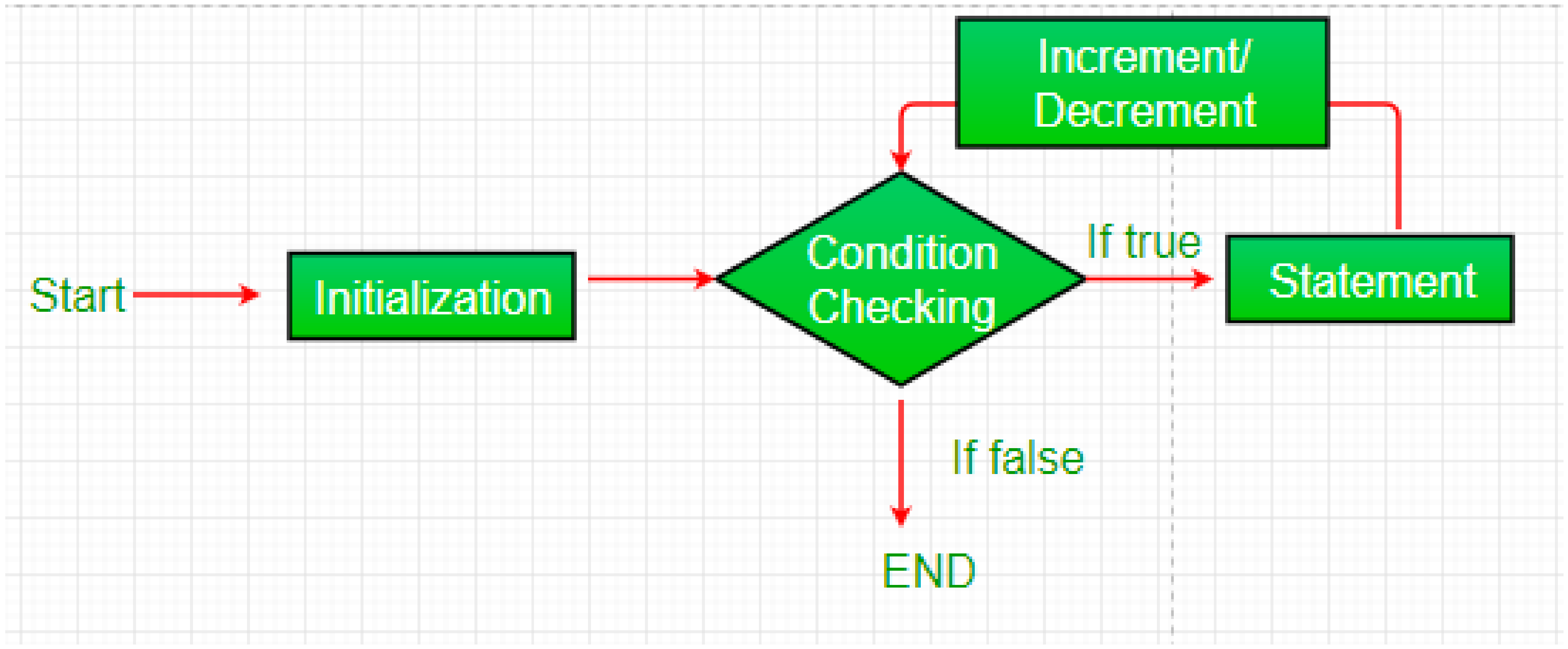
2. **Test Expression:** In this expression, we have to test the condition. If the condition evaluates to true then, we will execute the body of the loop and go to update expression. Otherwise, we will exit from the for loop.

Example: `i <= 10`

3. **Update Expression**: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example: `i++;`

Flow chart for loop (For Control Flow):



- **Example : this program will print numbers from 1 to 10**

```
class forLoopDemo
{
    public static void main(String args[])
    {
        // Writing a for loop

          for (int i = 1; i <= 5; i++)
          {
                    System.out.println(i);
          }
    }
}
```

STATIC MEMBERS

- Static is a keyword.
- It is a modifier.
- Any member of a class is prefixed with static modifier, then it is known as static member of a class.
- Static members are also known as class members.
- Static members are :
  - → static method
  - → static variable
  - → static initializers

- **Note** : static members can be prefixed only for a class member(ie.. Members declared in a class.

STATIC METHOD

- A method prefixed with the static modifier is known as static method

- **Characteristics :**
  - → static method block is stored in method area and reference is stored in class static area.
  - → We can use static methods with or without creating objects of the class.
  - → We can use the static methods with the help of class name as well as with the help of object reference.
  - A static method of the class can be used in any class with the help of class name.

- Example :

```
class Demo
{
        public static void test()
        {
                System.out.println(“hello”);
        }
        public static void main(String[  ]args)
        {
                test();
                Demo.test();
        }
}
```

Output :

```
hello
hello
```

- **Within a static context,**
    - → static members can be called directly with its name.
    - → non static members cannot be directly called with its name.
    - → this keyword cannot be used in here.

- **Static context :**
    - The block which belongs to the static method and multi line static initializer is known as static context.

STATIC VARIABLE

- Variables declared in a class block and prefixed with the static modifier is known as static variable.

- Characteristics :
  - → it is a member of a class.
  - → it will b assigned with default values.
  - → memory will be allocated in class static area.
  - → It is global in nature, it can be used within the class as well as outside the class.
  - → We can use the static methods with the help of class name as well as with the help of object reference.
  - → static variables of the class can be used in any class with the help of class name.

- **Note :** If static variables and local variables are in same name, then we can differentiate static variables with the help of class name.

- **Example :**

```
class Demo
{
        static int a;
        static int b;
        public static void main(String [ ] args)
        {
                System.out.println(a);
                System.out.println(b);
                int a=10;
                System.out.println(a);
                System.out.println(Demo.a);

        }
```

Output :

```
0
0
10
0
```

STATIC INITIALIZERS

- **We have two types:**
    - → single line static initializers
    - → multi line static initializers

- **Single line static initializers :**
    - Syntax: static datatype variable = value;
    - Example :   static int a=10;

- **Multi line static initializers :**
    - Syntax :

```
static
{
        // statements;
}
```

Example :

```
static
{
        System.out.println("hii");
}
```

- Purpose of static initializers :

→Static initializers are used to execute the start up instructions.
→The static initializers gets executed before the actual execution of main method.

- Example :

```
class Demo
{
        static
        {
                System.out.println(“ Executing first“);
        }
        public static void main(String [ ] args)
        {
                System.out.println(“executing after initializer”);
        }
```

Output :

```
Executing first
Executing after initializer
```

CLASS LOADING PROCESS

- To execute a java program, a portion of memory in RAM is allocated for JRE(Java Runtime Environment).
- In that portion of memory allocated, we have different range of memory, hence they are classified into four types :

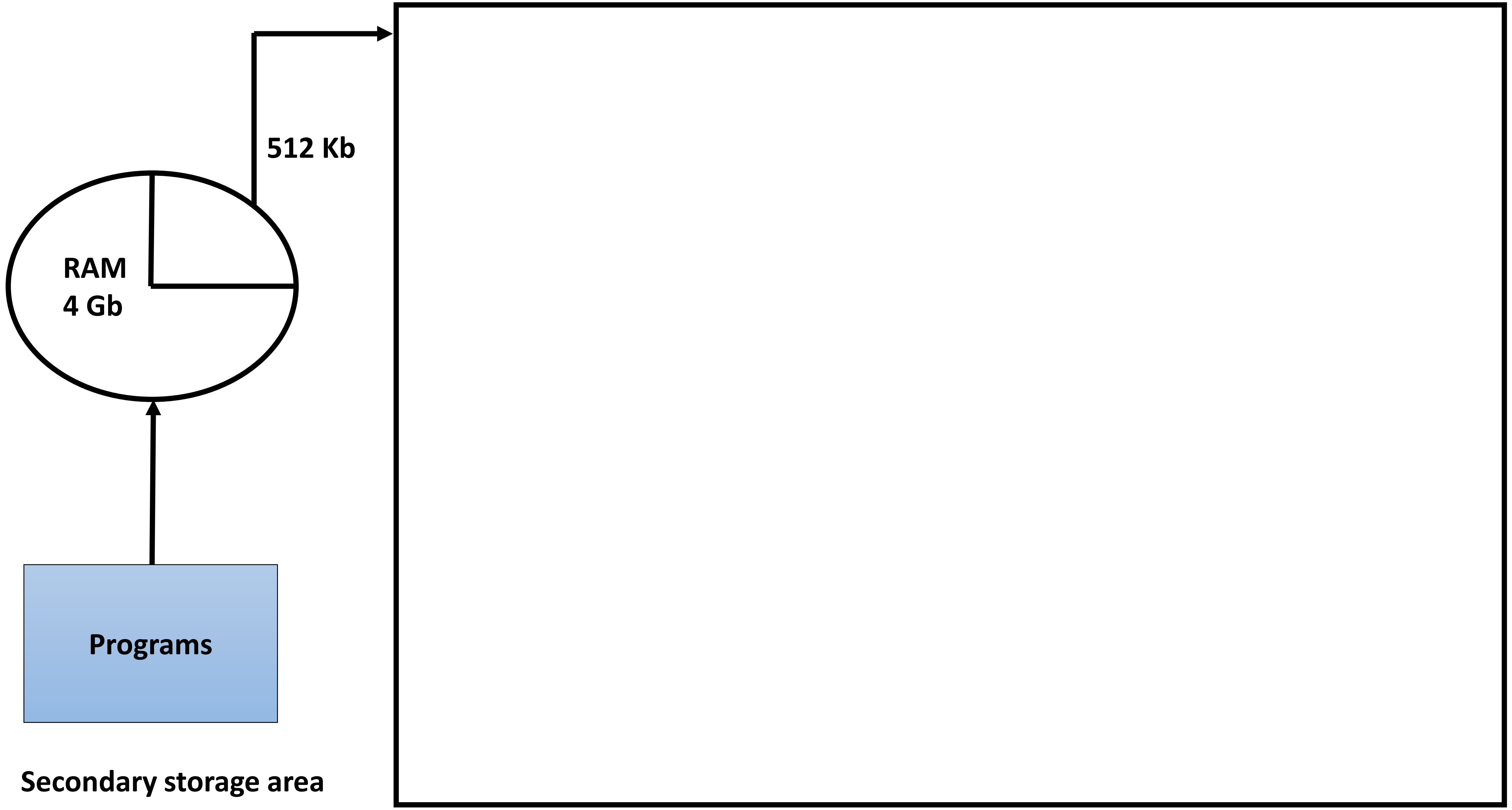
→ method area

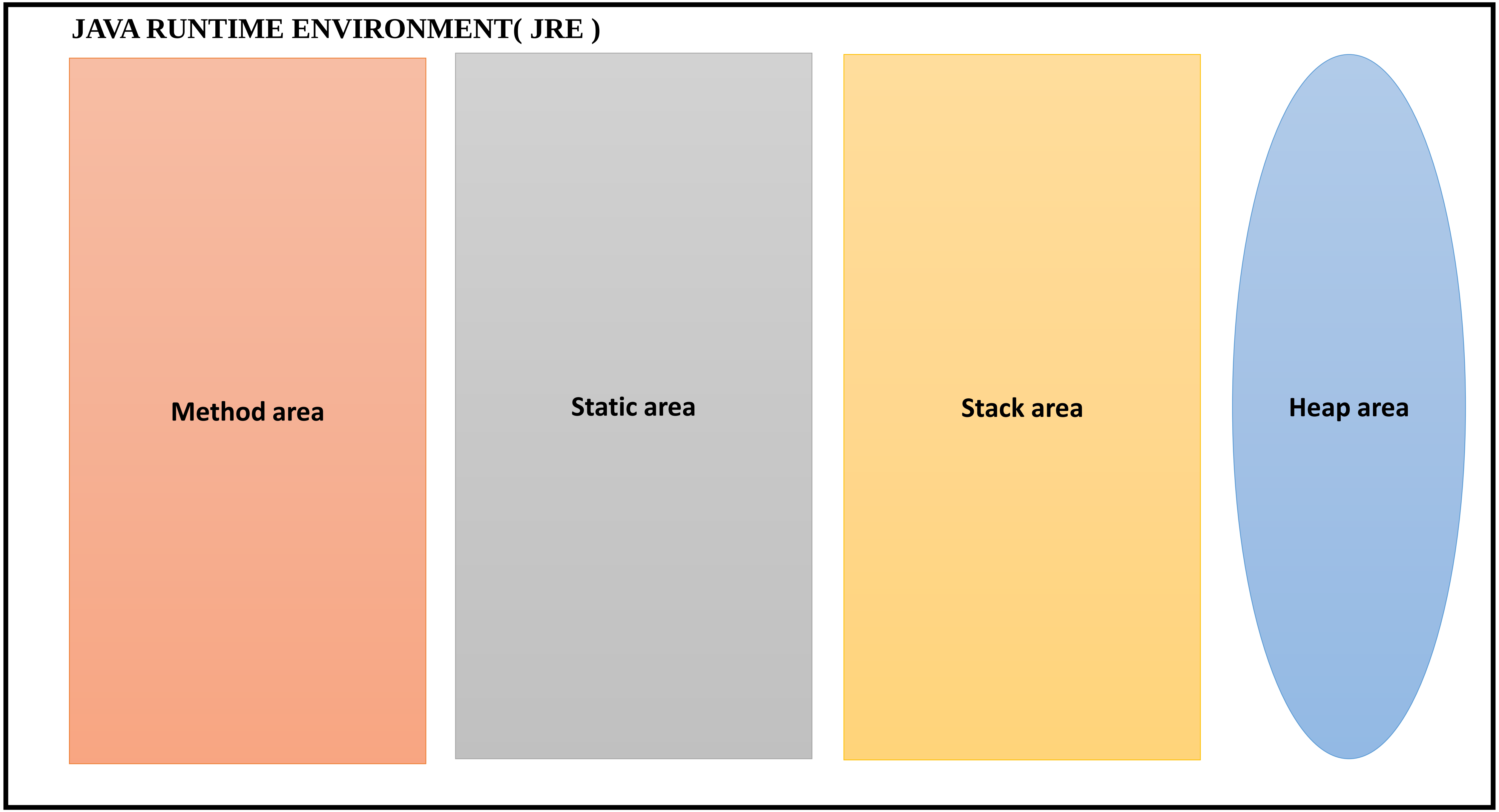
→ static area

→ stack area

→ heap area

JAVA RUNTIME ENVIRONMENT(JRE)




JAVA RUNTIME ENVIRONMENT(JRE)
Method area
Static area
Stack area
Heap area

- Method area – All the methods blocks and static multi line initializers will be stored in the method area.
- Class static area – for every class, a block of memory will be created in static area which is known as class static area.

The static members of the class will be allocated inside the memory created for the class.

- Stack area – used for execution of instructions.

For every method and multi line initializer that is under execution, a block of memory will be created in the stack area which is known as frames.

- Heap area – in heap area, a block of memory is created for object.

Every object can be accessed of a reference.

All the non static members of the class will be loaded inside block of memory in heap area.

- A block of memory is created in static area for the class which is loaded, which is known as class static area.
- All the methods (method definition) and multi line initializers will be loaded in method area and reference value will be assigned to each method block.
- All the static methods and static multi line initializers references are stored in class static area along with the method signature in the form of table.
- All the static variables are loaded into class static area and assigned with default values.
- All initializers are executed in top to bottom order.
- For every multi line static initializer, a frame is created in stack area. Once the execution is completed, the frame is removed.
- Once all the initializers are executed, we can say class loading process is complete.
- JVM will then call the main method for execution.
- A frame will be created in stack area for main method and execution of instructions begin line by line.
- For every method called, execution of main method is paused and a frame will be created in stack area for the method called.
- Once the execution of called method is completed, the frame is removed and control comes back to calling method(caller).
- The process continues. Once all instructions in main method are executed, the main method frame is removed.
- We can say execution is complete.

OBJECTS

Objects :

Any thing which has existence in real world is known as an object.

Every object will have attributes and behaviour.

objects in java :

-- According to OOPs, object is a block of memory created in heap area during runtime. It represents the real world object.

-- An object consists of attributes and behaviour. Attributes are represented with the help of non static variables. Behaviour is represented with the help of non static methods.

CLASS :

-- Before creating an object, the blueprint of the object must be designed , which provides the specification of the object. This is done with the help of class.

-- A class is a user defined non primitive data type. It represents the blueprint of the real world object.

-- A class provides specification of real world objects.

Note : We can create any number of objects for a class. Objects are also called as instance of a class.

STEPS TO CREATE AN OBJECT :

-- create a class or use an existing class which is already created.

-- Instantiation

INSTANTIATION :

-- the process of creating an object is known as instantiation.

Syntax to create an object :

```
Classname variablename = new Classname( );
```

new :

-- new is a keyword.

-- It is an unary operator.

-- It is used to create a block of memory in heap area during runtime and call the constructor.

-- Once the object is created, the new keyword will return the reference of that object.

CONSTRUCTOR :

-- It is a special member of the class whose name is same as the class name.

-- Constructor is used to load all the non static members of class into the object created.

Example :

```
class Employee
{
        String ename;
        int eid;
        String branch;
        public static void main(String [ ] args)
        {
                Employee e = new Employee ( );
                System.out.println(e.ename);
                System.out.println(e.eid);
                System.out.println(e.branch);
        }
}
```

NON STATIC MEMBERS

- Non static variables
- Non static methods
- Non static initializers
- Constructors

NON STATIC VARIABLES :

- A variable declared inside a class block and not prefixed with a static modifier is known as non static variable.

- **CHARACTERISTICS :**

- We cannot use the non static variable without creating an object.
- We can only use the non static variable with the help of object reference.
- Non static variables are assigned with default values during the object loading process.
- Multiple copies of non static variables will be created (Once for each object).

NON STATIC METHODS

- A method declared inside a class block and not prefixed with a static modifier is known as non static method.

- **CHARACTERISTICS :**

- All non static methods will get stored inside the method area and reference of these non static methods will be stored inside the object created in heap area.
- We cannot call the non static methods without creating an object of the class.
- We cannot access the non static methods directly with the class name.

Non static context :

- The block which belongs to the non static method and multi line non static initializer is known as non static context.

- Inside a non static context, we can use static and non static members of the class directly by using its name.

- Inside a static context, Non static members cannot be accessed directly with their names .

NON STATIC INITIALIZER

- It will execute during the object loading process.
- It will execute once for every object of a class created.

- **Purpose of non static initializers :**

- Non static initializers are used to execute the start up instructions for an object.

- **Types of non static initializers :**

- Single line non static initializer
- Multi line non static initializer

- **Single line static initializers :**
  - Syntax: datatype variable = value;
  - Example : int a=10;
- **Multi line static initializers :**
  - Syntax :

```
                {
                        // statements;
                }
Example  :
                {
                        System.out.println(“hii”);
                }
```

- **Example** :

```
class Demo
{
          {
                    System.out.println(“ Executing first“);
          }

          {
                     System.out.println(“ Executing second”);
          }
          public static void main(String [ ] args)
          {
                    System.out.println(“executing main method”);
                    Demo d = new Demo ();
          }
}
```

Output :

```
Executing main method
Executing first
Executing second
```

OBJECT LOADING PROCESS EXAMPLE :

EXAMPLE :

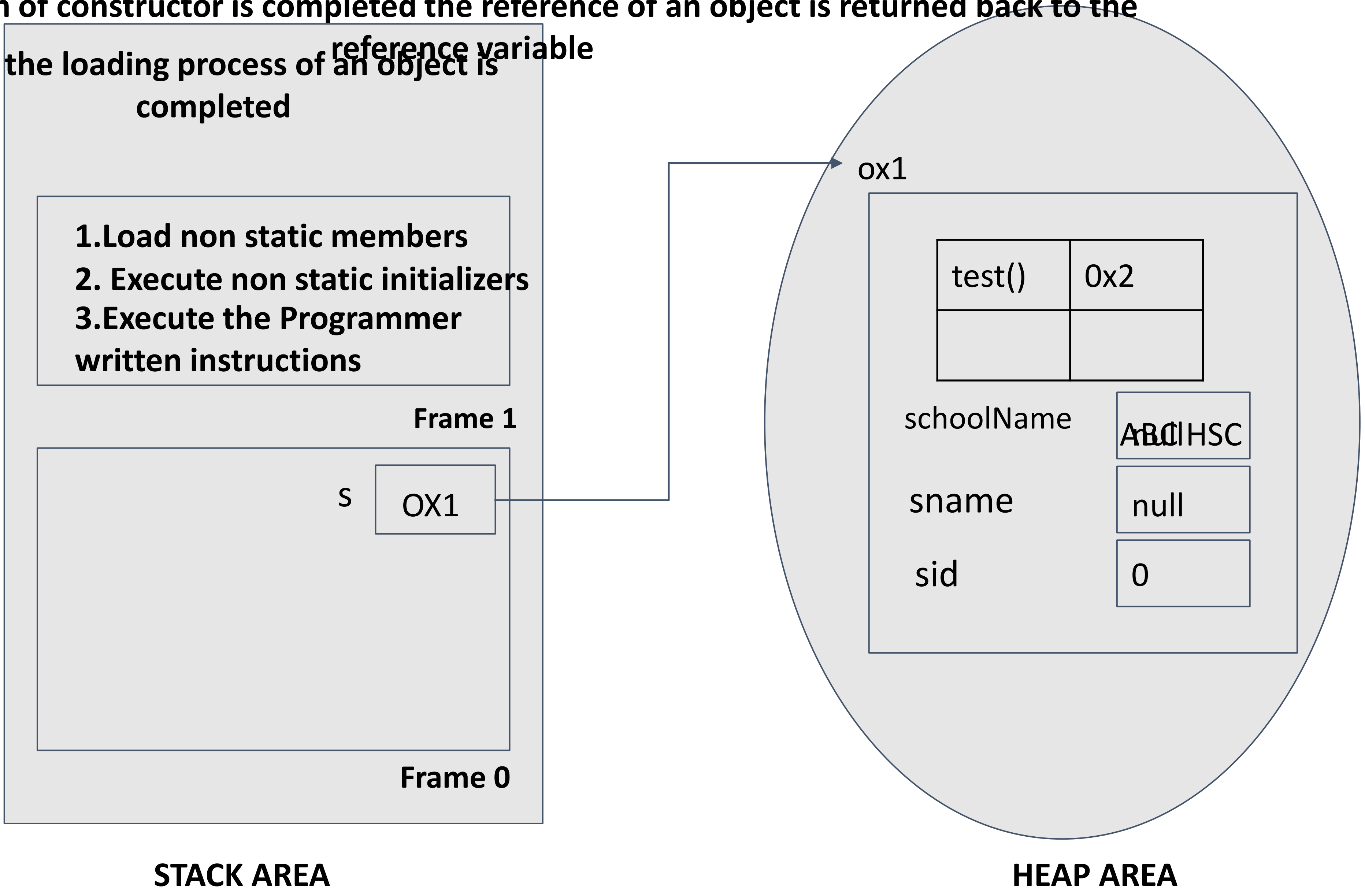
```
class School
{
        String schoolName = “ABC HSC”;
        {
                        S.o.pln(“School name is “+ schoolName);
        }
        public void test()
        {
                        S.o.pln(“From test”);
        }
        String sname;
        Int sid;
        public static void main(String []args)
        {
                School s = new School();
        }
}
```

New keyword will create a block of memory in a heap area.

Constructor is called

Once execution of constructor is completed the reference of an object is returned back to the reference variable

Thus the loading process of an object is completed


1.Load non static members
2. Execute non static initializers
3.Execute the Programmer written instructions

THIS KEYWORD :

- It is a keyword.
- It is a non static variable which holds the reference of the currently executing object.

- **Uses of this keyword :**
- It is used to access the members of current object.
- It is used to give the reference of the current object.
- Reference of a current object can be passed down from a method using this keyword.
- Calling a constructor of the same class is achieved with the help of this keyword.

CONSTRUCTOR

Constructor :

- A constructor is a special type of non static method whose name is same as the class name but it does not have a return type.

- Syntax :

```
[ access modifier ] [ modifier ] classname ( [ formal arguments ] )
{
    // Initialization
}
```

<u>CONSTRUCTOR BODY :</u>

- A constructor body will have following things :
  - → Load all the non static members into the object created.
  - → Non static initializers of the class are executed.
  - → Programmer written instructions are executed.

<u>PURPOSE OF CONSTRUCTOR :</u>

- During the execution of constructor,
  - →Non static members of the class will be loaded into the object created.
  - →If there is a non static initializers in the class, they are executed from top to bottom order.
  - →Programmer written instructions of the constructor gets executed.

<u>Note</u> : If the programmer fails to create a constructor, then the compiler will add a default no argument constructor.

CLASSIFICATION OF CONSTRUCTOR :

Constructors can be classified into 2 types based on formal arguments,

1. No argument constructor : A constructor which doesn’t have a formal argument is known as no argument constructor.

2. Parameterized argument : A constructor which consists of formal arguments is known as parameterized constructor.

NO ARGUMENT CONSTRUCTOR :

Syntax :

```
        [ access modifier ] [ modifier ] classname ( )
        {
                // instructions;
        }
```

Note : If the programmer fails to create a constructor, compiler will add a no argument constructor implicitly.

• **Example :**

```
class Student
{
            String name;
            int id;
            Student  ( )
            {
                        System.out.println(“ no argument constructor “):
            }
            public static void main(String [ ]  args)
            {
                        Student s = new Student ( );
                        s.name = “ abc “;
                        s.id = 10;
                        System.out.println( “name : “+name+” id : “+id );
            }
}
```

Output :

```
abc
10
```

PARAMETERIZED CONSTRUCTOR :

- The constructor which has formal arguments is called as parameterized constructor.
- The purpose of parameterized constructor : They are used to initialize the non static variables by accepting the data from the constructor in the object loading process.

Syntax :

```
[ access modifier ] [ modifier ] classname ( formal arguments )
{
        // Initialization ;
}
```

- **Example :**

```
class Student
{
        String name;
        int id;
        Student  (String name, int id )
        {
                this.name=name;
                this.id=id;
        }
        public static void main(String [ ]  args)
        {
                Student s = new Student ( “ Ram “, 10);
                System.out.println(s.name);
                System.out.println(s.id);
        }
}
```

Output :

```
Ram
10
```

LOADING PROCESS OF AN OBJECT

- New keyword will create a block of memory in heap area.
- It then calls the constructor.
- During the execution of constructor,

    → All non static members of the class are loaded into the object.
    → If there are non static initializers, they are executed from top to bottom order.
    → Programmer written instructions of the constructor will be executed.

- The execution of the constructor is completed.
- The object is created successfully.
- The reference of the object created is returned by the new keyword.
- These steps are repeated for every object creation.

CONSTRUCTOR OVERLOADING

- If a class is having more than one constructor , it is known as constructor overloading.
- Rule : The signature of constructors must be different.

<u>Example :</u>

```
class Student
{
		String sname;
		int sid;
		Student ( )
		{
		}
		Student(String sname)
		{
				this.sname=sname;
		}
		Student(int sid)
		{
				this.sid=sid;
		}
		Student(String sname, int sid)
		{
				this.sname=sname;
				this.sid=sid;
		}
}
```

Creating objects for Student class :

Student s = new Student (“Seeta “ , 4);

Student s = new Student (“Seeta “);

Student s = new Student (4);

Student s = new Student ( );

CONSTRUCTOR CHAINING

- A constructor calling another constructor is known as constructor chaining.

- In java, we can achieve constructor chaining by using 2 ways :

→ this ( )

→ super ( )

this ( ) statement :

- It is used to call the constructor from another constructor within the same class.
- **Rule :**

1. this ( ) can be used only inside the constructor.
2. It should always be the first statement in the constructor.
3. The recursive call to the constructor is not allowed.
4. If a class has n constructors, it can have n-1 this ( ) statements.

NOTE : If a constructor has this ( ) statement , then the compiler doesn't add load instructions and non static initializers into the constructor body.

Example :

```
class Student
{
        String sname;
        int sid;
        long cno;
        Student ( )
        {
        }
        Student(String sname)
        {
                this.sname=sname;
        }
        Student(String sname, int sid )
        {
                this(sname);
                this.sid=sid;
        }
        Student(String sname, int sid, long cno)
        {
                this(sname, sid);
                this.cno=cno;
        }
}
```

Creating an object for Student class :

Student s = new Student (“Seeta “ , 4 , 123456789);

PRINCIPLES OF OOPS

OOPs has following principles :

1. **ENCAPSULATION**
2. **INHERITANCE**
3. **POLYMORPHISM**
4. **ABSTRACTION**

ENCAPSULATION

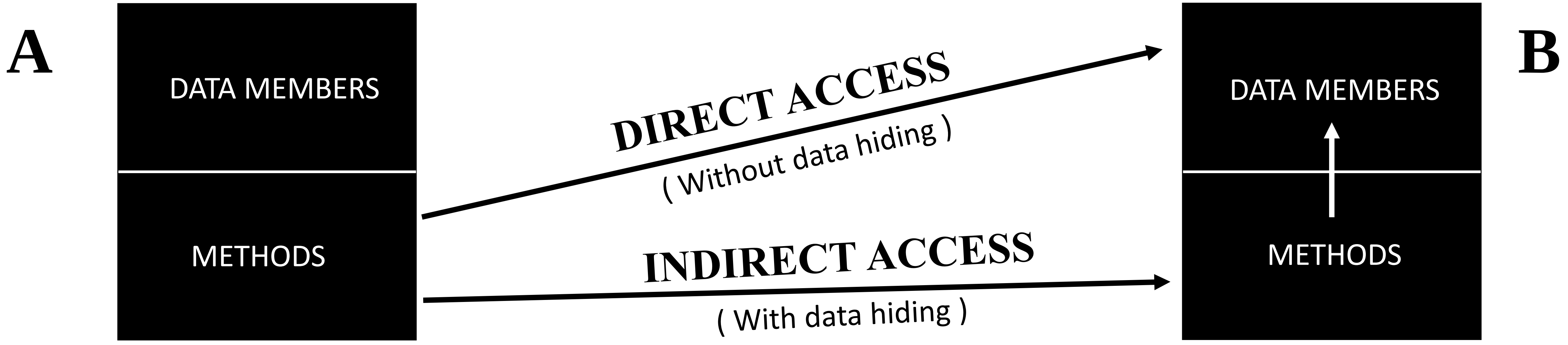
- **Encapsulation** **:** The process of binding the state (Attributes) and behaviours of an object together is known as encapsulation.

- We can achieve encapsulation in java with the help of class. A class has both state and behaviour of an object.

- **Advantage** : By using encapsulation, we can achieve data hiding.

- **Data Hiding** : It is a process of restricting the direct access of data members of an object and providing indirect secured access of data members via methods (public) of same object is known as data hiding.

- Data hiding helps to achieve verification and validation of data before storing and modifying it.



- With data hiding, direct access is not possible, only indirect access is allowed.
- Without data hiding, direct access is possible, also indirect access is possible.

- **Steps to achieve data hiding :**

1. Prefix the data members of the class with private modifier.
2. Design a getter and setter method.

PRIVATE MODIFIER :

- private is an access modifier.
- It is a class level modifier.
- If the members of the class are prefixed with private modifier, then we can access those members only within the class.

NOTE : Data hiding can be achieved with the help of private modifier.

- **Example :**

```
class Book
{
        private int book_id;
        private String book_name;
        private double price;

        Book ( int book_id, String book_name, double price)
        {
                this.book_id=book_id;
                this.book_name=book_name;
                this.price=price;
        }
        public static void main(String[] args)
        {
                Book b=new Book(1214, " JAVA ", 450.0);
                System.out.println ( "book id : "+book_id+"book name " +book_name+" price " +price);
        }
}
```

GETTER AND SETTER METHODS

Getter method :

- It is used to fetch/get the data .
- The return type of getter method is the type of hidden value.

Setter method :

- It is used to update or set the data.
- The return type is always void.

- **NOTE :** The validation and verification can be done in setter method before storing data and in getter method before reading the private data members.

- If you want to make your hidden value only readable, then create only getter method.

- If you want to make your hidden value only modifiable, then create only setter method.

- If you want to make your hidden value both readable and modifiable , then create both getter and setter method.

- If you want to make your hidden value neither readable nor modifiable, then don’t create a getter and setter method.

ADVANTAGES OF DATA HIDING :

- Provides security to the data members.
- We can verify and validate the data before modifying it.
- We can make the data members of the class to
  - → only readable
  - → only modifiable
  - → both readable and modifiable
  - → neither readable nor modifiable

IS-A RELATIONSHIP

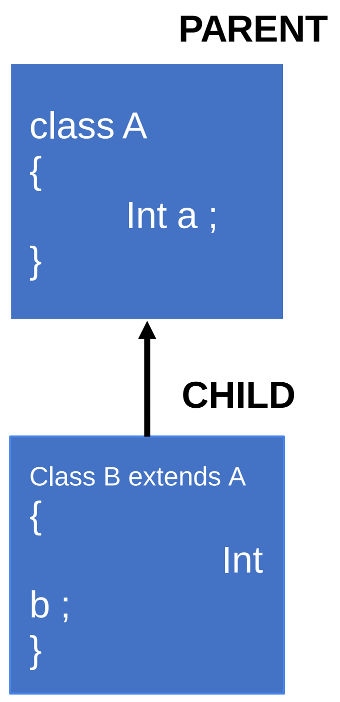
Is-A RELATIONSHIP :

- The relationship between two objects which is similar to the parent and child relation is known as the Is-A relationship.
- In an Is-A relationship, the child object will acquire all properties of the parent object, and child object will have its own extra properties.
- In an Is-A relationship, we can achieve generalization and specialization.

NOTE :

- **Parent is called generalized.**
- **Child is called specialized.**

EXAMPLE :



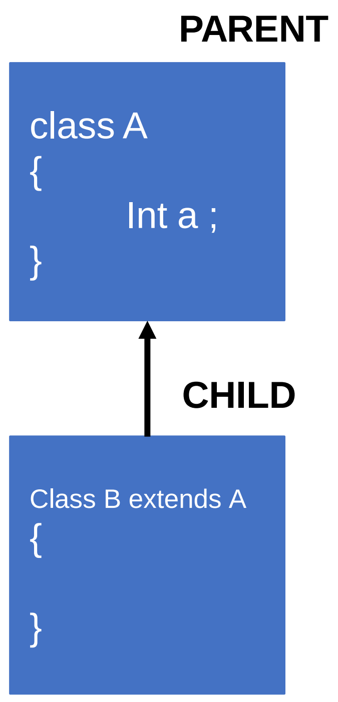
```
class A
{
        Int a ;
}
```

```
Class B extends A
{
                Int
b ;
}
```

DECLARED 1

INHERITED 1
DECLARED 1

TOTAL 1 + 1 = 2



```
class A
{
        Int a ;
}
```

```
Class B extends A
{

}
```

DECLARED 1

INHERITED 1
DECLARED 0

TOTAL 1 + 0 = 1

- With the help of the child class reference type, we can use the members of the parent's class as well as the child.
- With the help of parent class reference, we can use only the members of a parent but not the child class.

TERMINOLOGIES

PARENT CLASS :

The parent class is also known as a superclass or base class.

CHILD CLASS :

The child class is also known as a subclass or derived class.

NOTE : Is-A relationship is achieved with the help of inheritance.

INHERITANCE :

The process of one class acquiring all the properties and behavior from the other class is called inheritance.

In java, we can achieve inheritance with the help of

1. **extends keyword**
2. **implements keyword**

EXTENDS KEYWORD

extends keyword :

extends keyword is used to achieve inheritance between two classes.

EXAMPLE :

```
class A
{
        Int i = 10;
}
Class B extends A
{
        Int j = 20;
        public static void main(String argos[])
        {
                B b = new B();
                S.o.pln( b.i ); //CTS
                S.o.pln( b.j ); //CTS
                A a = new A();
                S.o.pln( a.i );  //CTS
                S.o.pln( a.i )  //CTE with the help of parent variable we can only access parent members
        }
}
```

SESSION 1 ACTIVITY

EXAMPLE PROGRAM 1

Step1: Create a class A.

Step2: Declare and initialize Non-static variables.

Step3: Create a method display() to print - Hello from A.

Step4: Create a class B and make it a child class for A.

Step5: Declare and initialize Non-static variables.

Step6: create an object for class B.

Step7: Call the display() method of class A and print the values in the variables with the reference variable of class B.

EXAMPLE PROGRAM 2

Step1: Create a class A.

Step2: Declare Non-static variables.

Step3: Create a method display() to print - Hello from A.

Step4: Create a class B and make it a child class for A.

Step5: Declare Non-static variables.

Step6: create an object for class B.

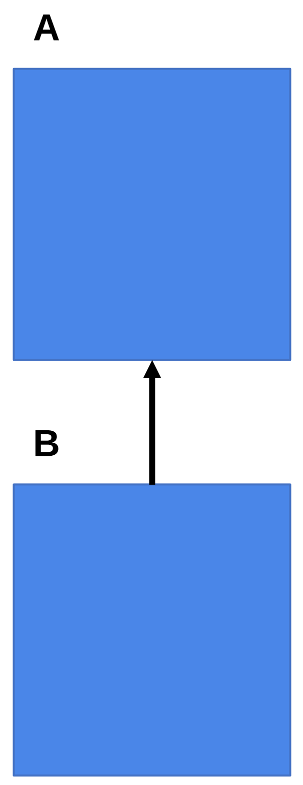
Step7: Initialize all the variables.

Step8:Call the display() method of class A and print the values in the variables with the reference variable of class B.

Types of inheritance

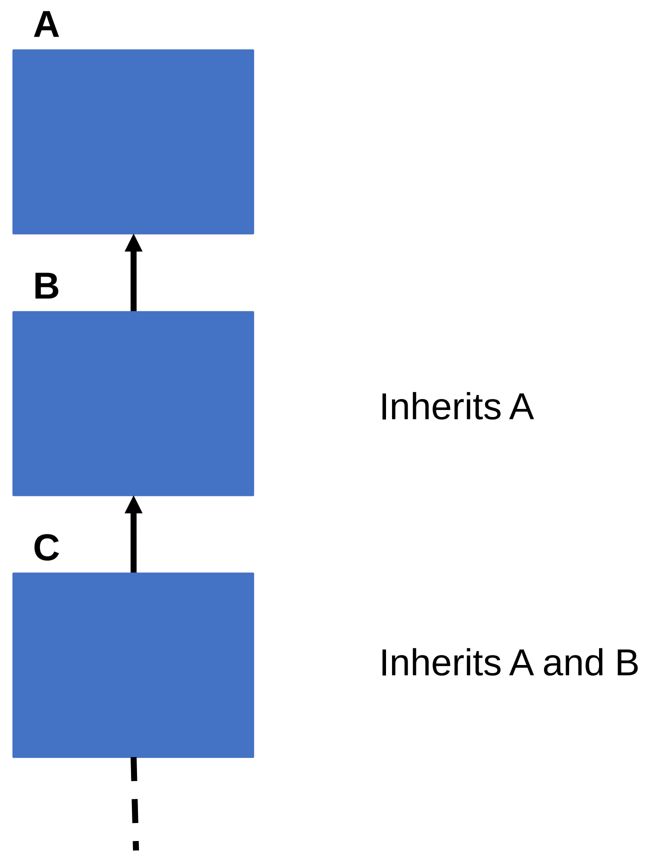
1.SINGLE LEVEL INHERITANCE :

Inheritance of only one level is known as single-level inheritance.



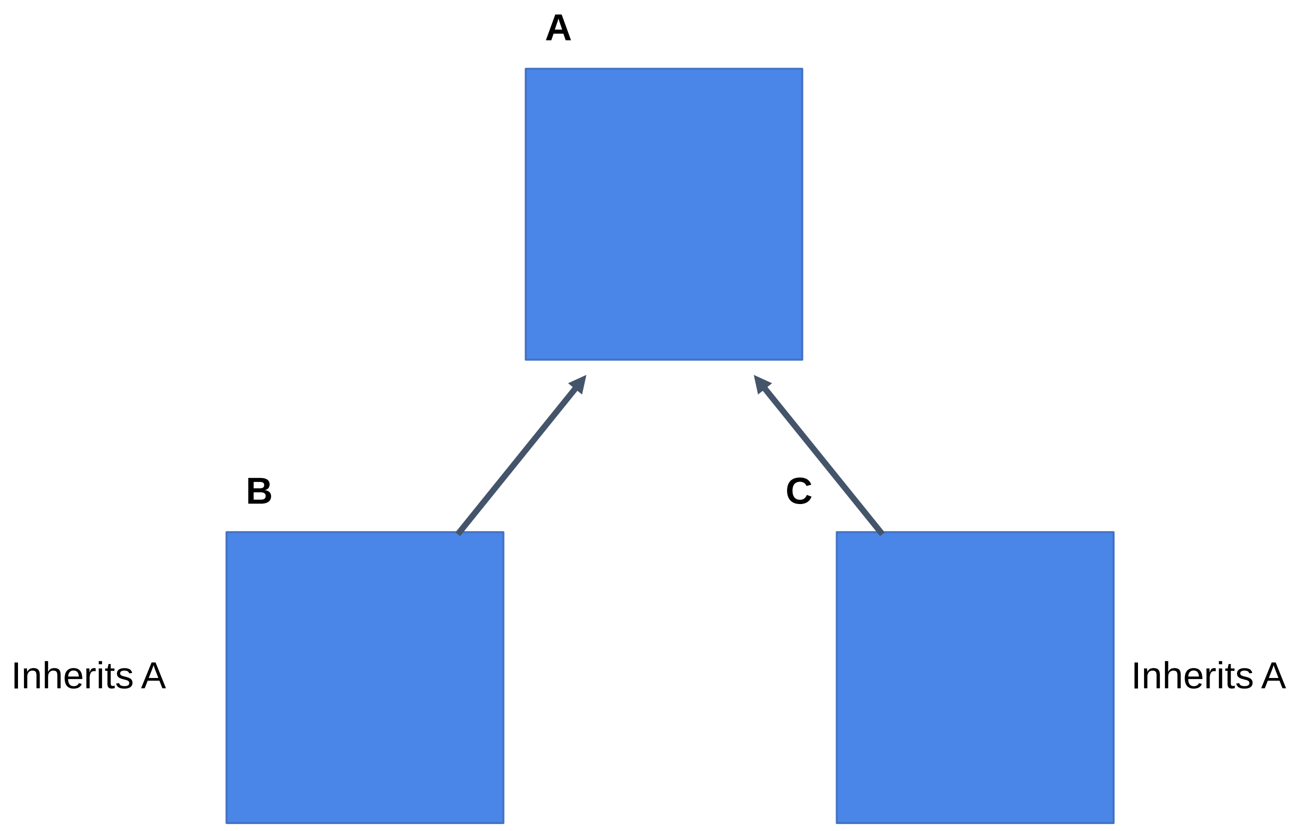
2. MULTI LEVEL INHERITANCE :

Inheritance of more than one level is known as multi-level inheritance.



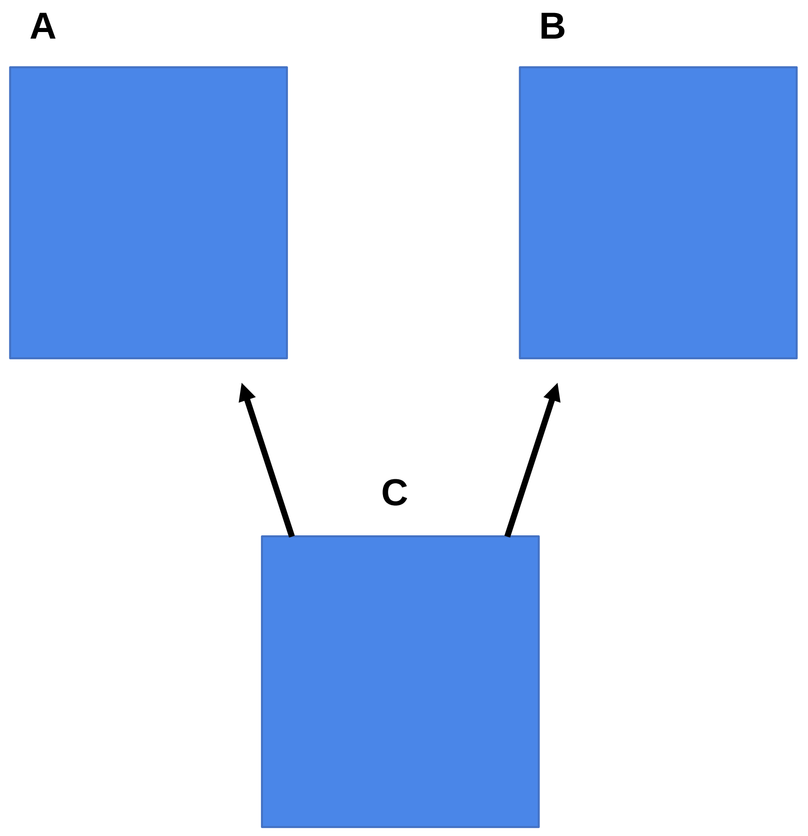
3. HIERARCHICAL INHERITANCE :

If a parent (superclass) has more than one child (subclass) at the same level then it is known as hierarchical inheritance.



MULTIPLE INHERITANCE :

If a subclass (child) has more than one parent (Super class) then it is known as multiple inheritance.



NOTE :

- **Multiple inheritance has a problem known as the diamond problem.**
- **Because of the diamond problem, we can’t achieve multiple inheritance only with the help of class.**
- **In java, we can achieve multiple inheritance with the help of an interface.**

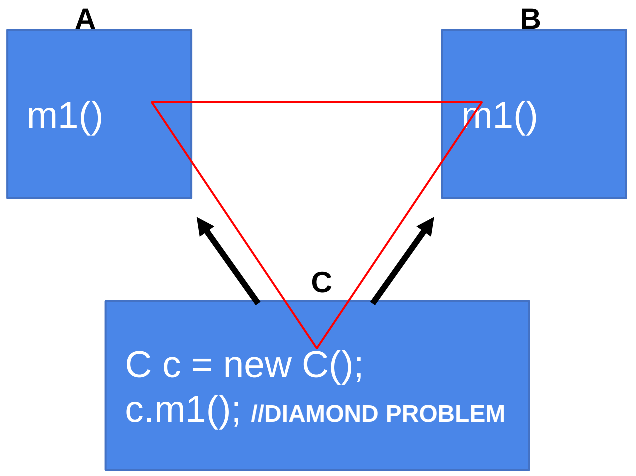
DIAMOND PROBLEM

DIAMOND PROBLEM :

Assume that there are two classes A and B. Both are having the method with the same signature. If class C inherits A and B then these two methods are inherited to C .

Now an ambiguity arises when we try to call the super class method with the the help of subclass reference.

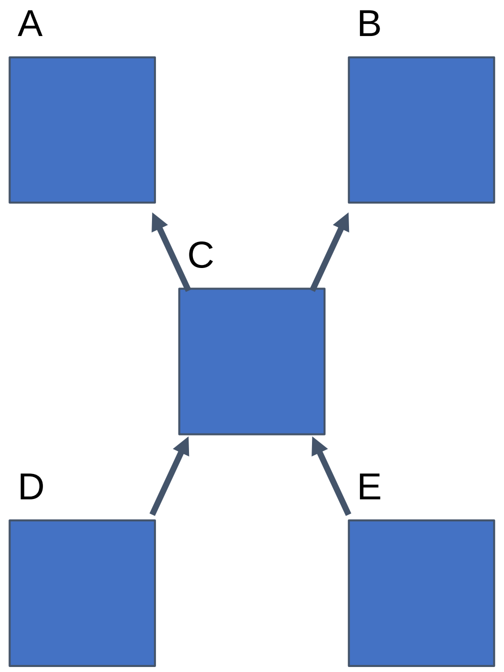
This problem is known as the **diamond problem.**

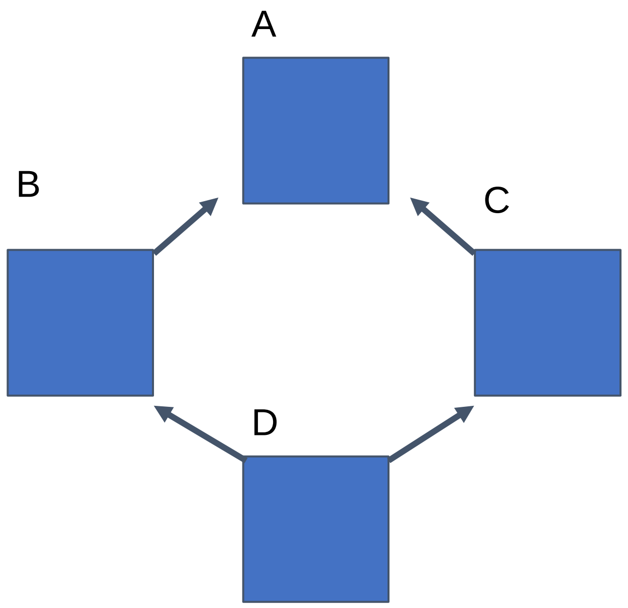


HYBRID INHERITANCE

HYBRID INHERITANCE :

The combination of multiple inheritance and hierarchical inheritance is known as hybrid inheritance.





EXAMPLE PROGRAM 1

Step1: Create a class Author.

Step2: create three Non-static variables (authorName, age and place)

Step4: Create a class Book make it a child class for Author.

Step5: create two Non-static variables (name and price).

Step6: Create a parameterized constructor and initialize (name, price and author).

Step7: Create a main method.

Step8: Create object for Book class.

Step9: Print the values present in variables (name, price, authorName, age and place).

SUPER() CALL STATEMENT

super() CALL STATEMENT :

- super is a keyword, it is used to access the members of the superclass.
- super() call statement is used to call the constructor of the parent class from the child class constructor.

PURPOSE OF SUPER() STATEMENT :

- When the object is created, the super call statement helps to load the nonstatic members of the parent class into the child object.
- We can also use the super() call statement to pass the data from the subclass to the parent class.

RULE TO USE SUPER() STATEMENT

RULE TO USE SUPER() STATEMENT

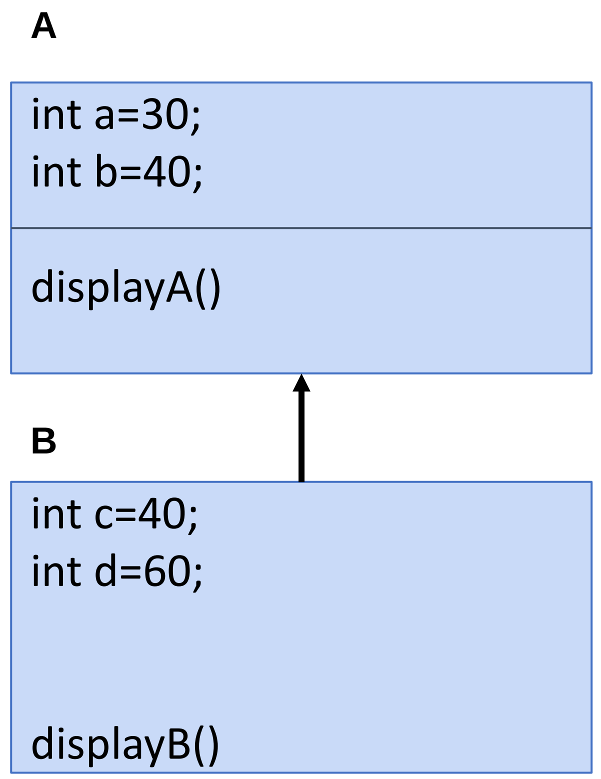
- super() call statement should always be the first instruction in the constructor call.
- If a programmer doesn't use the super() call statement, then the compiler will add a no-argument super call statement into the constructor body.

DIFFERENCE BETWEEN THIS() AND SUPER()

this()	**super()**
It is used to call the constructor of the same class	It is used to call the constructor of the parent class(Super class)
It is used to represent the instance of child class	It is used to represent the instance of parent class
It should be used as a first statement in a constructor block	It should be used as a first statement in a constructor block

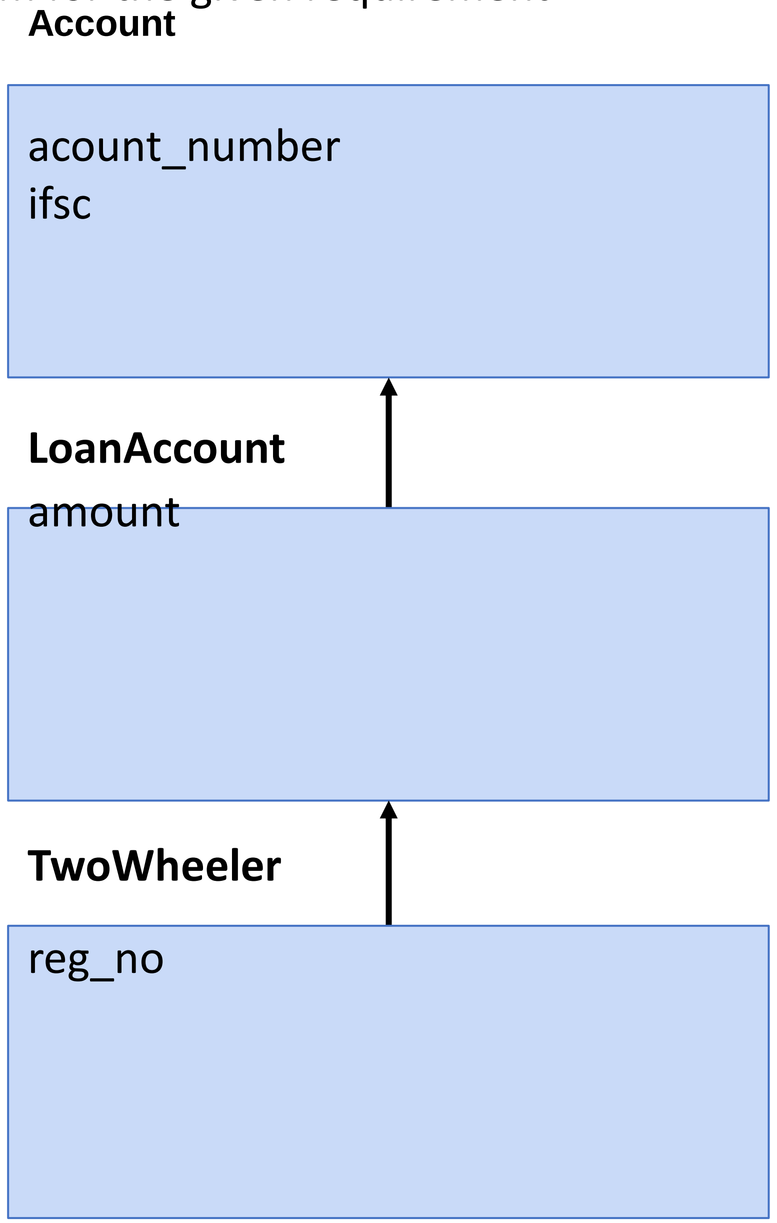
ASSIGNMENT

Write a java program for the given requirement



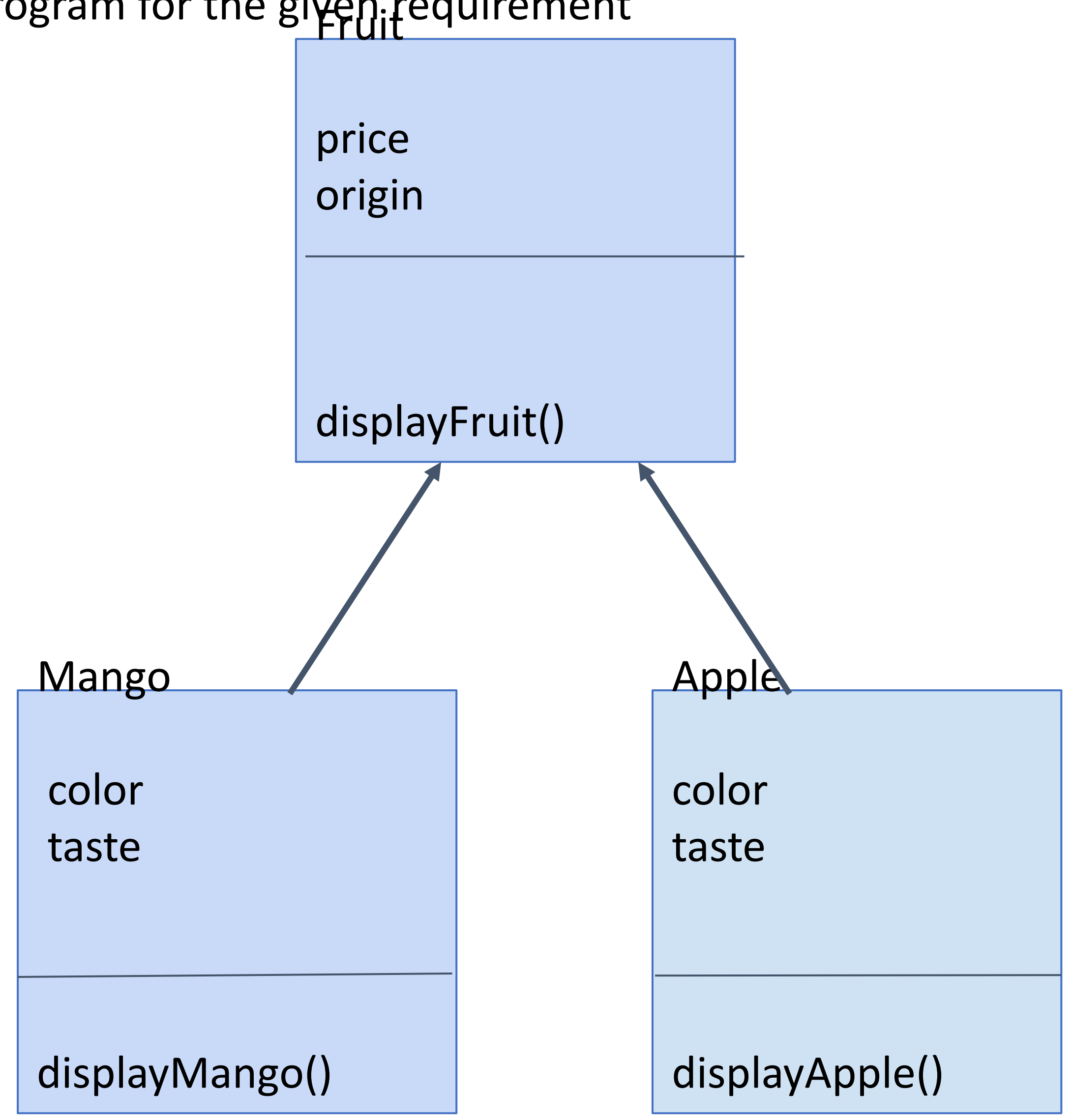
Display the values in the variables using reference variable of class B

Write a java program for the given requirement



Initialize and Display the values in the variables using reference variable of class

Write a java program for the given requirement



- Display the values in the variables using reference variable of class Mango
- Display the values in the variables using reference variable of class Apple

UNDERSTANDING REFERENCE VARIABLE

UNDERSTANDING REFERENCE VARIABLE :

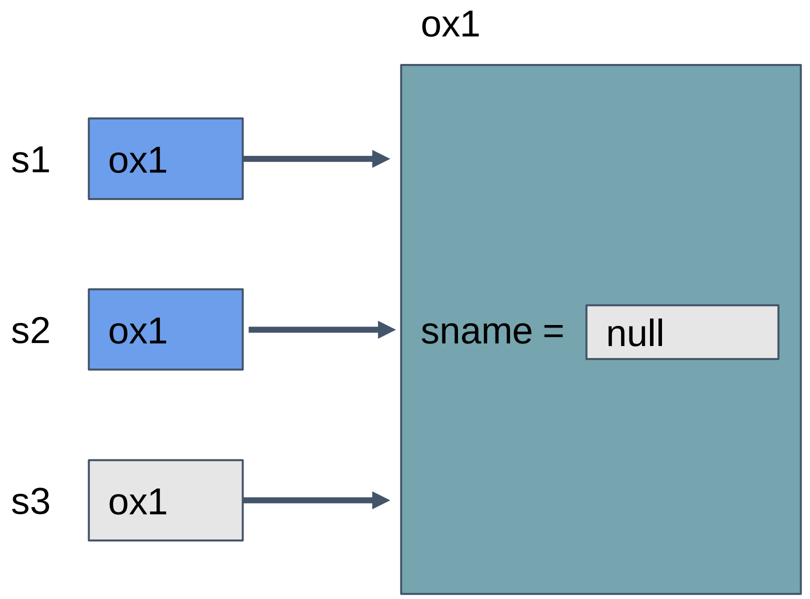
- One object can be referred to with multiple reference variables.
- We can copy the reference of one object into multiple reference variables.

EXAMPLE :

```
class Student
{
        String sname ;
}

Student s1 = new Student();
Student s2 = s1 ;

Student s3 = s1 ;
```



NOTE :

We can access the members of the Student object by using s1 or s2 or s3. [The state of the object can be modified by using any of the reference variables.]

NOTE :

- **We can copy the reference from one variable to another variable only if both the reference variable are the same type.**
- **If both the reference variable are different types then conversion of one reference type to another is required.**

NON PRIMITIVE TYPE CASTING

NON PRIMITIVE TYPE-CASTING (DERIVED TYPE CASTING)

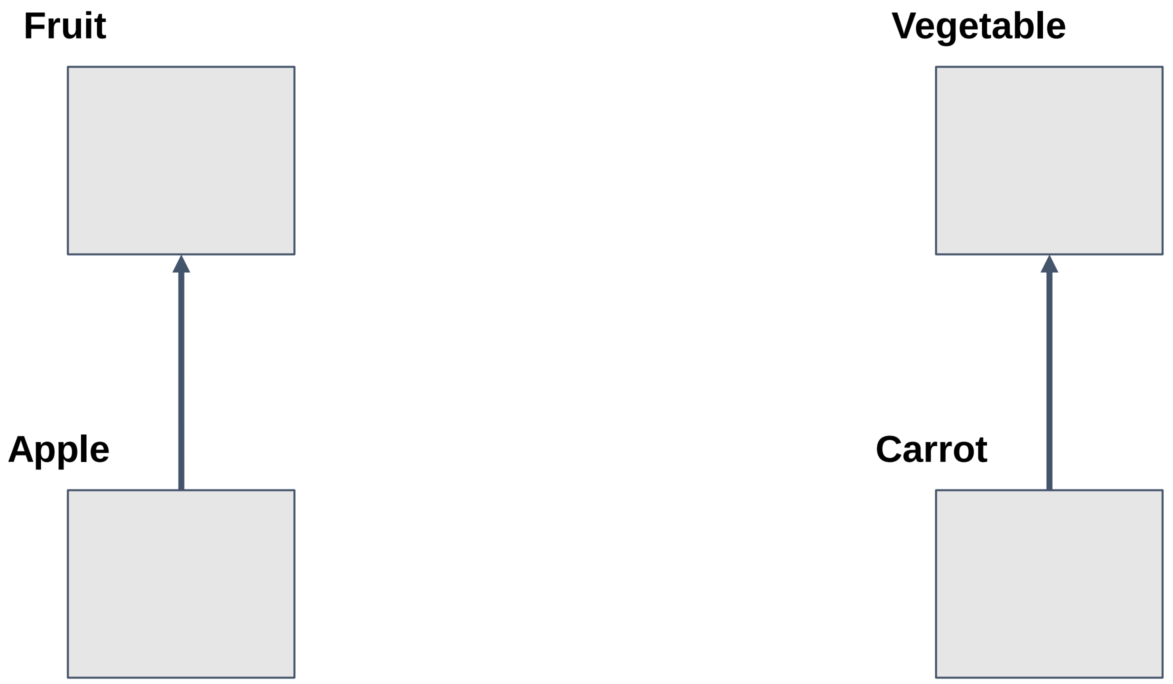
The process of converting one reference type into another reference type is known as non-primitive or derived type casting.

RULES TO ACHIEVE NON PRIMITIVE TYPE CASTING :

We can convert one reference type into another reference type only if it satisfies the following condition,

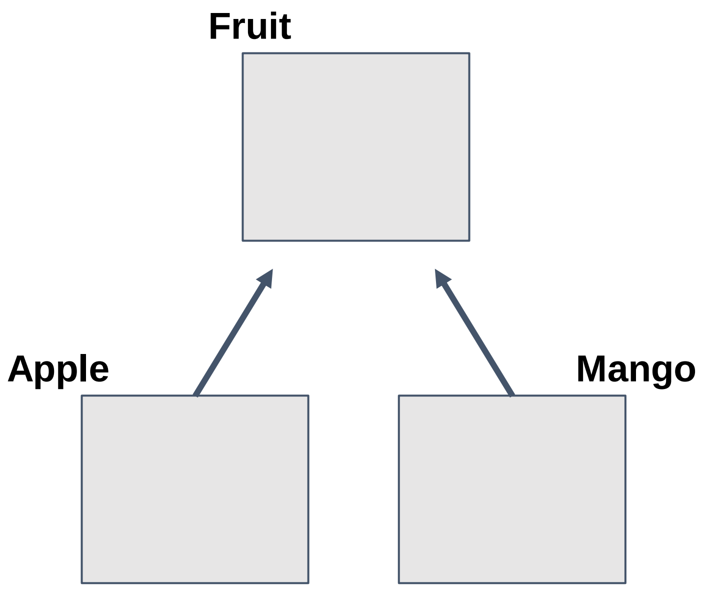
- There must be an Is-A relation (Parent and child) exist between two references.
- If the class has a common child.

EXAMPLE 1 :



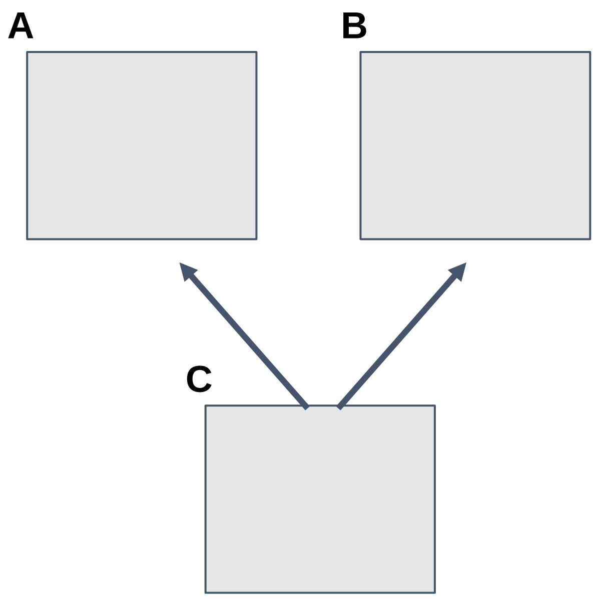
- Fruit can be converted to Apple and Apple can be converted to fruit as well as Vegetable can be converted to Carrot and Carrot can be converted to Vegetable.
- But Fruit and apple can’t be converted to Vegetable and carrot as well as Vegetable and carrot can’t be converted to Fruit and Apple.

EXAMPLE 2 :



- Fruit can be converted to Apple as well as Mango and Mango and Apple can be converted into Fruit.
- But Apple can’t be converted into Mango as well as Mango can’t be converted into Apple**.**

EXAMPLE 3 :



We can convert A to B type , B to A type , A to C type , C to A type , B to C type and C to B type.

TYPES OF DERIVED TYPE CASTING

TYPES OF NON PRIMITIVE OR DERIVED TYPE CASTING :

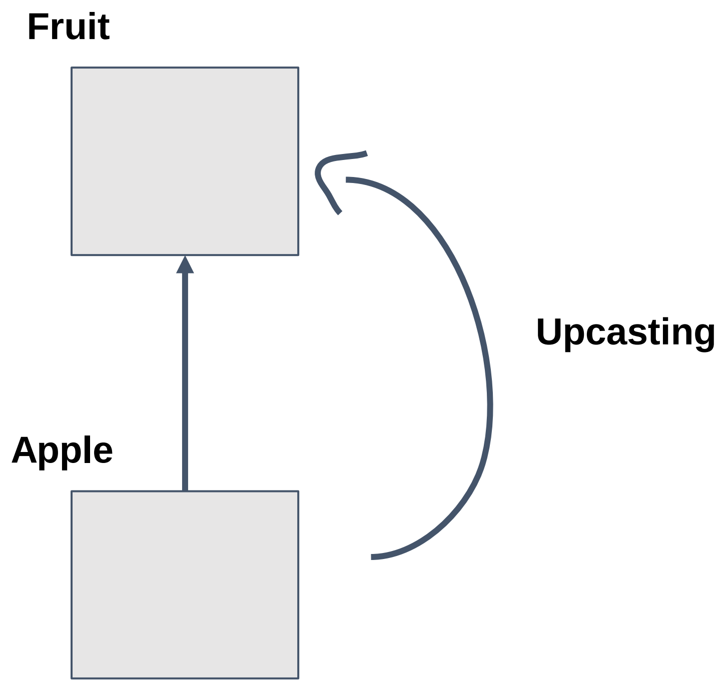
Non primitive type casting can be further classified into two types,

I. **Upcasting**
II. **Downcasting**

UPCASTING

UPCASTING :

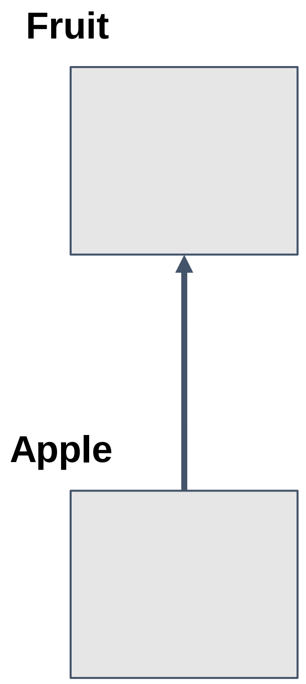
The process of converting the child class reference into a parent class reference type is known as upcasting.



NOTE :

- The upcasting is implicitly done by the compiler.
- It is also known as auto upcasting.
- Upcasting can also be done explicitly with the help of a typecast operator.
- Once the reference is upcasted we can’t access the members of the child.

EXAMPLE :



```
Apple a = new Apple();
Fruit f = a; // Upcasting
```

NOTE :

By using ‘f’ we can access only the members of aFruit (Super class)

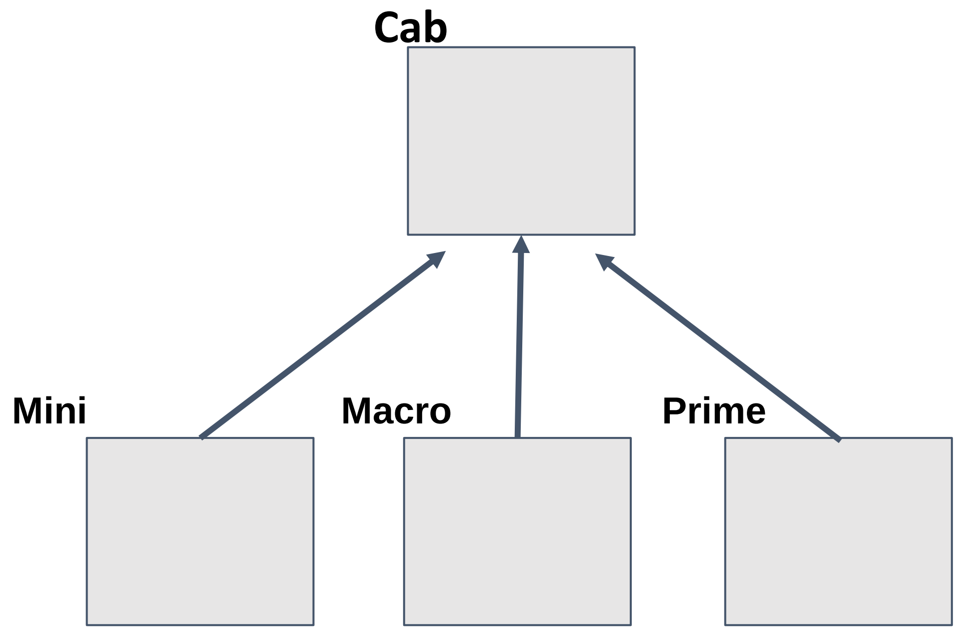
WHY DO WE NEED UPCASTING ?

- It is used to achieve generalization.
- It helps to create a generalized container so that the reference of any type of child object can be stored.

EXAMPLE :

Generalized variable

```
Cab c ;
c = new Mini();
c = new Macro();
c = new Prime();
```



DISADVANTAGE

DISADVANTAGE :

There is only one disadvantage of upcasting that is, once the reference is upcasted its child members can’t be used.

NOTE :

In order to overcome this problem, we should go for downcasting

EXAMPLE PROGRAM 1

Step1: Create a class Fruit.

Step2: create a variable fruit and initialize it.

Step3: create a class Vegetable.

Step4: create a vegetable variable and initialize it.

Step5: Create a class Driver.

Step6: create a main method.

Step7: create an object for the Class Fruit and store the reference in a variable of type Fruit.

Step8: create an object for the Class Vegetable and store the reference in a variable of type Vegetable.

Step9: create an object for the Class Vegetable and store the reference in a variable of type Fruit.

Step10: create an object for the Class Fruit and store the reference in a variable of type Vegetable.

EXAMPLE PROGRAM 2

Step1: Create a class Parent.

Step2: create a variable p and initialize it.

Step3: Create a class Child and make it a child class for Parent.

Step4: create a variable c and initialize it.

Step5: Create a class Driver.

Step6: create a main method.

Step7: create an object for the Class Child and store the reference variable of type Child and print the values present in the variables p and c.

Step8: create an object for the Class Child and store the reference variable of type Parent and print the values present in the variables p and c.

DOWNCASTING

DOWNCASTING :

The process of converting a parent (superclass) reference type to a child (subclass) reference type is known as downcasting.

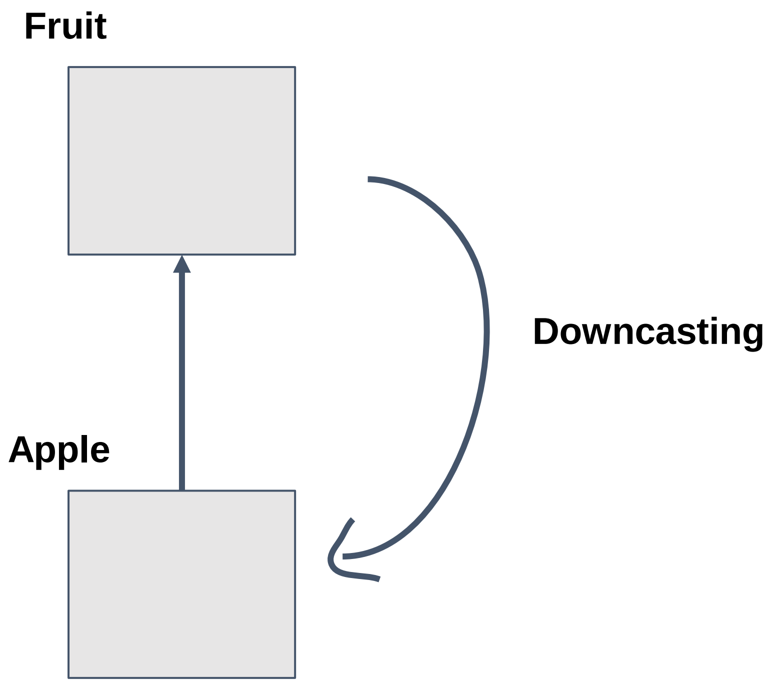
NOTE :

- **Downcasting is not implicitly done by the compiler.**
- **It should be done explicitly by the programmer with the help of a typecast operator.**

PURPOSE OF DOWNCASTING

WHY DO WE NEED A DOWNCASTING ?

- If the reference is upcasted, we can’t use the members of a subclass.
- To use the members of a subclass we need to downcast the reference to a subclass.



CLASS CAST EXCEPTION

ClassCastException :

- It is a RuntimeException.
- It is a problem that occurs during runtime while downcasting.

When and why do we get a ClassCastException?

When we try to convert a reference to a specific type(class), and the object does’t have an instance of that type then we get ClassCastException.

EXAMPLE:

```
Child c = (Child)new Parent(); //ClassCastException
```

instanceof operator

instanceof operator :

- It is a binary operator
- It is used to test if an object is of given type.
- The return type of this operator is boolean.
- If the specified object is of given type then this operator will return true else it return false.

Syntax to use instanceOf operator :

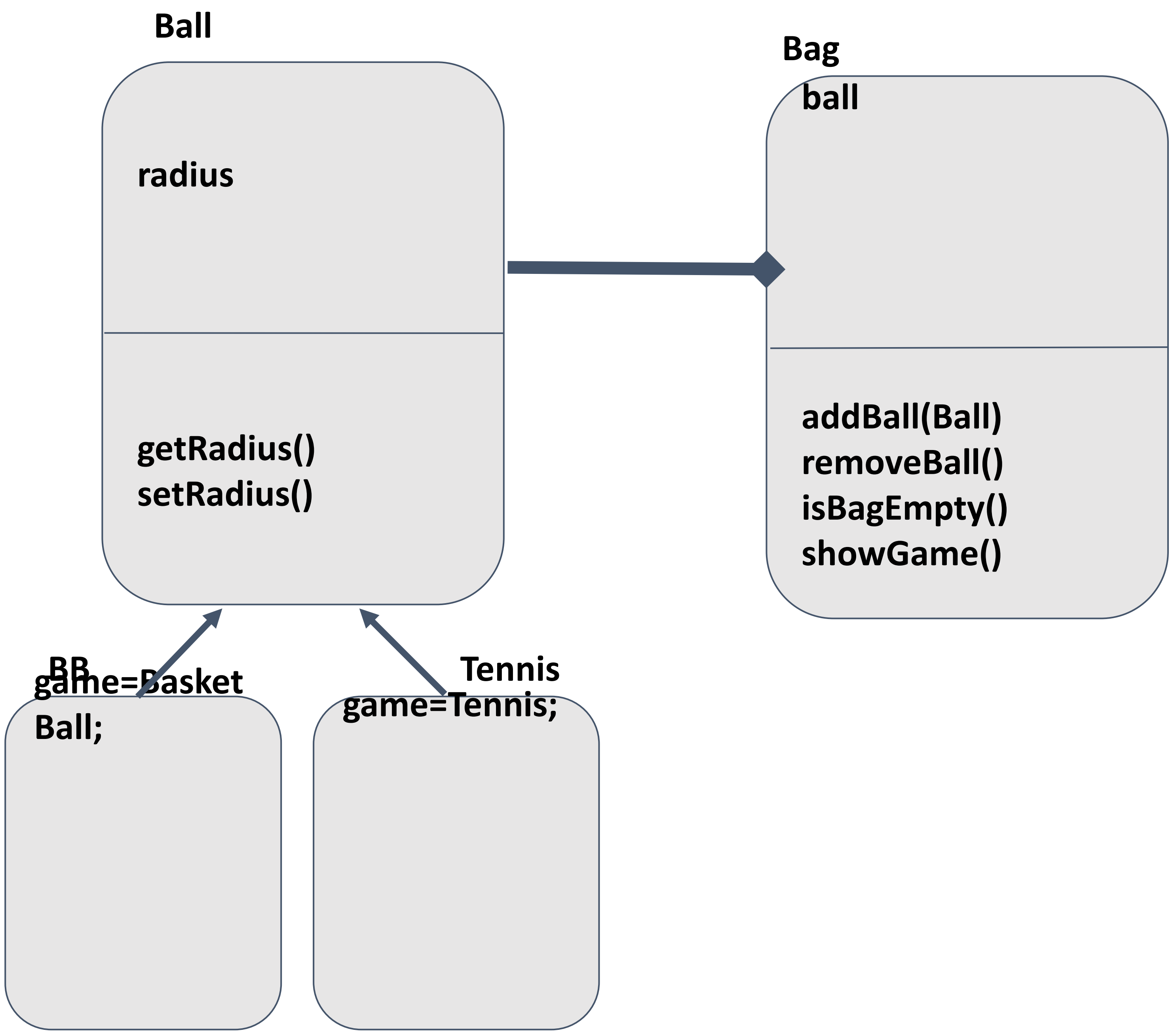
(Object_Ref) instanceof (type)

EXAMPLE : new String() instanceof Object ;

NOTE :

There must be Is-A relation exist between Object_Ref and type passed in an instanceof operator otherwise we will get a Compile time error.

PROGRAMS TO DEMONSTRATE THE USE OF UPCASTING AND DOWNCASTING



User Interface

S1 : create object of bag.

S2:Display Menu

1. Add Ball
2. Remove Ball
3. Check Bag is empty or not
4. Show game that can be played

S3:Read the choice.

POLYMORPHISM

POLYMORPHISM

Polymorphism is derived from two different Greek words ‘Poly’ means Numerous ‘Morphs’ means form Which means Numerous form. Polymorphism is the ability of an object to exhibit more than one form with the same name.

For Understanding :

One name	-----> Multiple forms
One variable name	------> Different values
One method name	-----> Different behaviour

TYPES OF POLYMORPHISM :

In java we have two types of polymorphism,

1. **Compile time polymorphism**
2. **Runtime Polymorphism**

COMPILE TIME POLYMORPHISM

COMPILE TIME POLYMORPHISM :

- If the binding is achieved at the compile-time and the same behavior is executed it is known as compile-time polymorphism.
- It is also said to be static polymorphism.
- It is achieved by

1. **Method overloading**
2. **constructor overloading**
3. **Variable shadowing**
4. **Method shadowing**
5. **Operator overloading (does not supports in java)**

METHOD OVERLOADING

METHOD OVERLOADING :

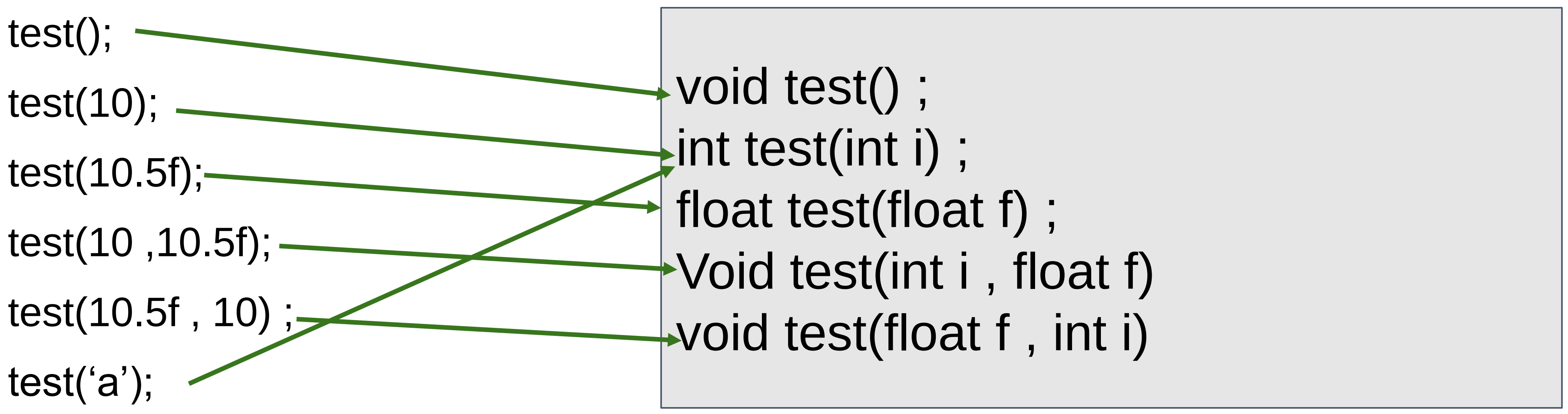
If more than one method is created with the same name but different formal arguments in the same class are known as method overloading.

EXAMPLE : java.lang.Math;

$$\left.\begin{array}{l}\textbf{abs(double d)}\\ \textbf{abs(float f)}\\ \textbf{abs(int i)}\\ \textbf{abs(long l)}\end{array}\right\}\textbf{Overloaded method}$$

These are some of the overloaded method (Method with same name but different formal arguments) implemented in java.lang.Math class.

EXAMPLE :



```
test();
test(10);
test(10.5f);
test(10 ,10.5f);
test(10.5f , 10) ;
test('a');
test(10 , 10);
```

```
void test() ;
int test(int i) ;
float test(float f) ;
Void test(int i , float f)
void test(float f , int i)
```

//CTE ambiguity

CONSTRUCTOR OVERLOADING

CONSTRUCTOR OVERLOADING :

A class having more than one constructor with different formal arguments is known as constructor overloading.

METHOD SHADOWING

METHOD SHADOWING :

If a subclass and superclass have the static method with same signature , it is known as method shadowing.

Which method implementation gets execute, depend on what ?

In method shadowing binding is done at compile time , hence it is compile time polymorphism. The execution of the method depends on the reference type and does not depend on the type of object created.

NOTE :

- **Return type should be same**
- **Access modifier should be same or higher visibility than super class method.**
- **Method shadowing is applicable only for the static method.**
- **It is compile time polymorphism**
- **Execution of implemented method depends on the reference type of an object.**

VARIABLE SHADOWING

VARIABLE SHADOWING :

If the superclass and subclass have variables with same name then it is known as variable shadowing.

Which variable is used, depend on what ?

In variable shadowing binding is done at compile time , hence it is a compile time polymorphism. Variable used depends on the reference type and does not depend on the type of object created.

NOTE :

- **It is applicable for both static and non static variable.**
- **It is a compile time polymorphism.**
- **Variable usage depends on type of reference and does not depend on type of object created.**

RUNTIME POLYMORPHISM

RUNTIME POLYMORPHISM :

- If the binding is achieved at the runtime then it is known as runtime polymorphism.
- It is also known as dynamic binding.
- It is achieved by method overriding.

METHOD OVERRIDING

METHOD OVERRIDING :

- If the subclass and superclass have the non static methods with and same signature , it is known as method overriding.

Rule to achieve method overriding :

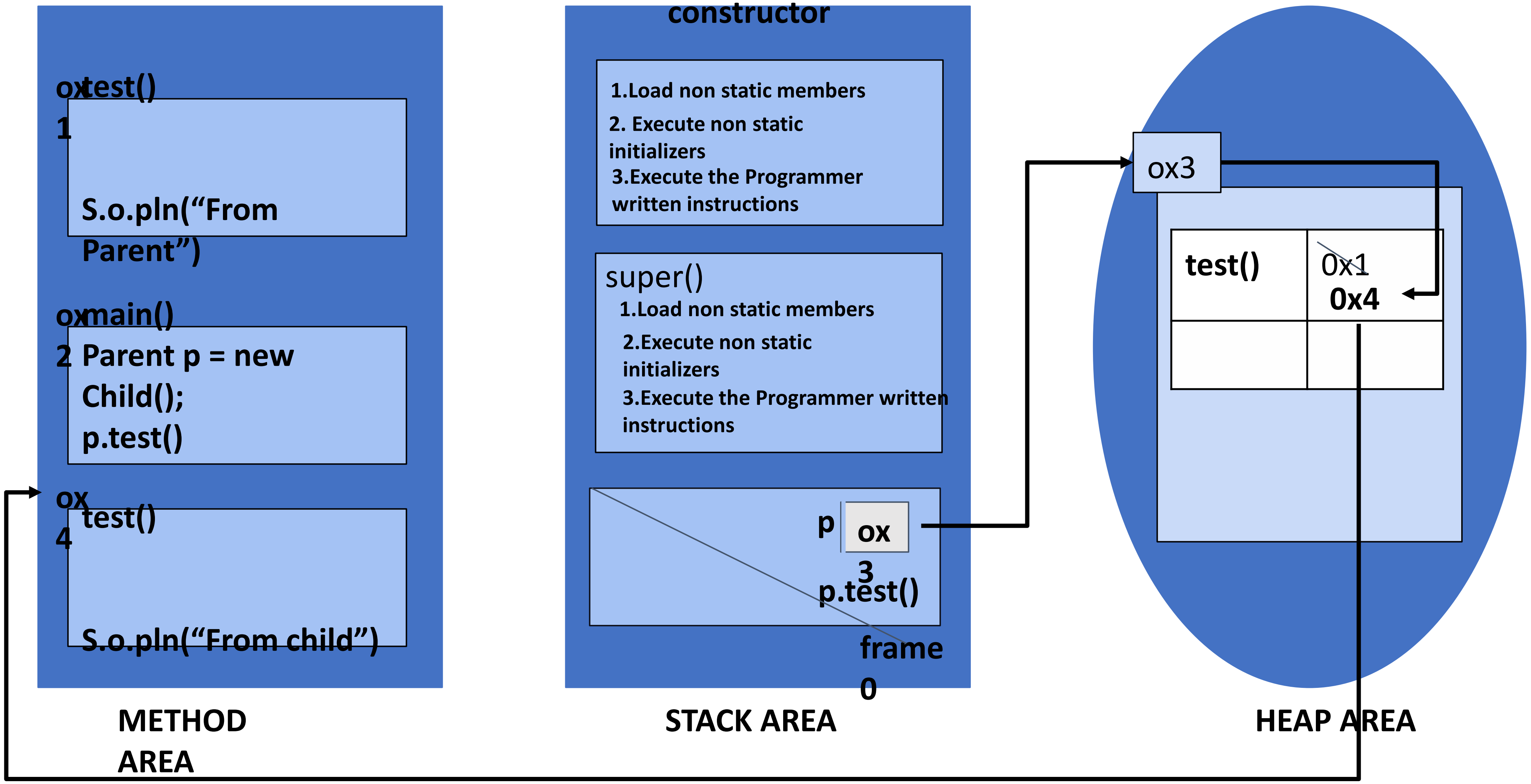
- **Is-A relationship is mandatory.**
- **It is applicable only for non static methods.**
- **The signature of the subclass method and superclass method should be same.**
- **The return type of the subclass and superclass method should be same until 1.4 version but, from 1.5 version covariant return type in overriding method is acceptable (subclass return type should be same or child to the parent class return type.).**

Method [illegible] static, new register is not created

[illegible] instead reference is replaced

[illegible] called by the Child class

constructor



ox1 test()
S.o.pln(“From Parent”)
ox2 main()
Parent p = new Child();
p.test()
ox4 test()
S.o.pln(“From child”)
1.Load non static members
2. Execute non static initializers
3.Execute the Programmer written instructions
super()
1.Load non static members
2.Execute non static initializers
3.Execute the Programmer written instructions
p ox3
p.test()
frame 0
ox3
test()
~~0x1~~
0x4
METHOD AREA
STACK AREA
HEAP AREA

EXAMPLE :

```
class Parent
{
    public void test()
    {
            System.out.println(“From parent”);
    }
}
class Child extends Parent
{
    @override
    public void test()
    {
            System.out.println(“From child”);
    }
    public static void main(String[] args)
    {
            Parent p = new Child();
            p.test();
    }
}
```

Why do we need method overriding ?

Method overriding is used to either modify or provide new design for an already existing inherited

EXAMPLE :

```
        class AppV1
`       {
                public void feature1()
                {
                        System.out.println(Oldest);
                }
        }
        class AppV2 extends AppV1
        {
                public void feature2()
                {
                        System.out.println(“New feature added”);
                }
                public void feature1()
                {
                        System.out.println(“Updated feature”);
                }
        }
Appv1 app = new Appv2();
```

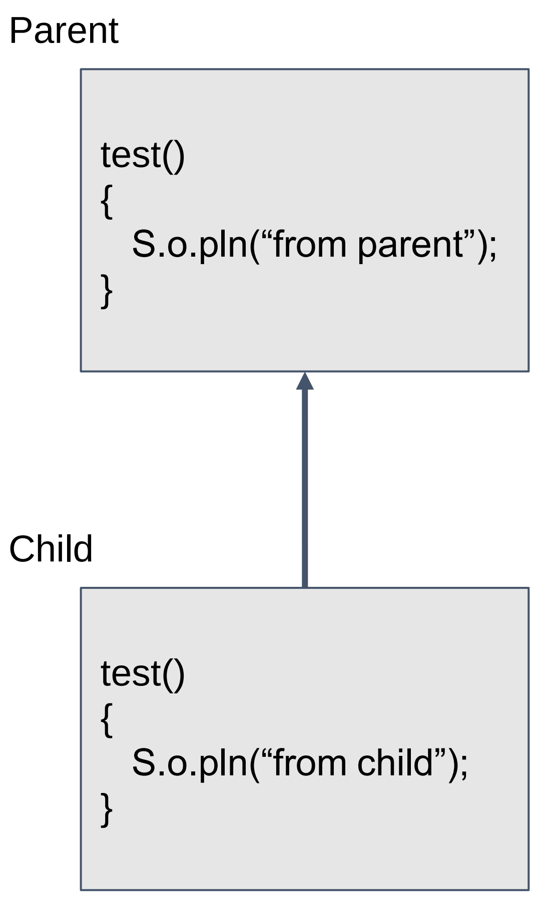
EXAMPLE :

```
Child c = new Child();
c.test(); // from child
Parent p = c;
p.test(); // from child
```

Internal runtime object is child so child test() will get executed, it does not depend on the reference type.

NOTE :

Variable overriding is not applicable.



FINAL MODIFIER

FINAL MODIFIER :

- final modifier indicates that an object is fixed and can’t be changed.
- It is applicable to variables , method , and class level object.

FINAL VARIABLE

FINAL VARIABLE :

If a variable is prefixed with final modifier then the value of the variable remains constant once it gets initialized.

NOTE :

If we try to change / modify the value of final variable then we will get compile time error.

EXAMPLE :

```
final int i = 10 ;
i = 20 ; //CTE
final int j ;
j = 30 ; // initializatIon
j = 40 ; //CTE
```

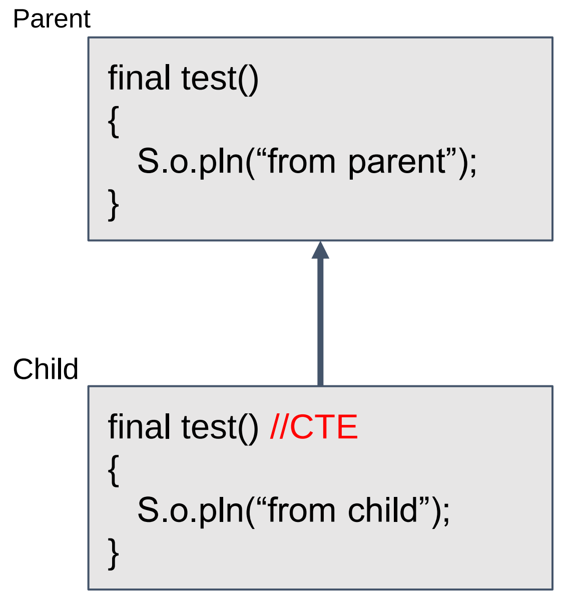
FINAL METHOD

FINAL METHOD:

If a method if prefixed with final modifier then overriding of that method is not possible.

NOTE :

If we try to override that method we will get compile time error.



```
final test()
{
  S.o.pln("from parent");
}
```

```
final test() //CTE
{
  S.o.pln("from child");
}
```

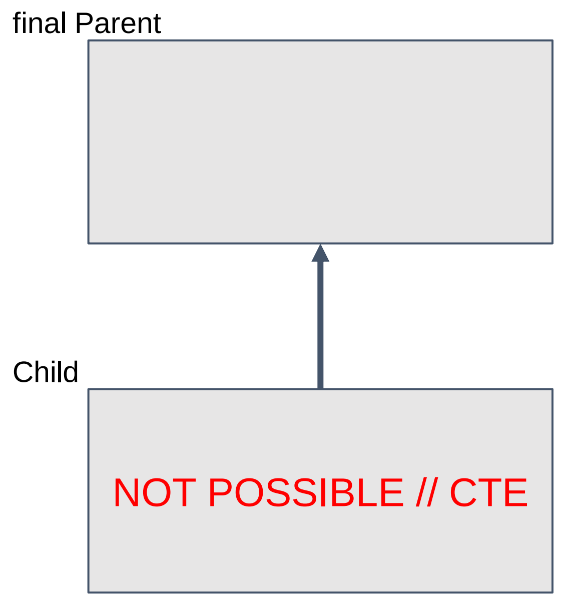
FINAL CLASS

FINAL CLASS :

If the class is prefixed with final modifier ithen we can’t inherit from that class is not possible).

NOTE :

If we try to create a child class for the final class then we will get compile time error.



INTRODUCTION TO OBJECT CLASS

Object class :

- Object class is defined in java.lang package.
- Object class is a supermost parent class for all the classes in java.
- In object class there are 11 non static methods.

METHODS IN OBJECT CLASS

1	public String toString()
2	public boolean equals(Object o)
3	public int hashCode()
4	protected Object clone() throws CloneNotSupportedException
5	protected void finalize()
6	final public void wait() throws InterruptedException
7	final public void wait(long l) throws InterruptedException
8	final public void wait(long l , int i) throws InterruptedException
9	final public void notify() throws InterruptedException
10	final public void notifyAll() throws InterruptedException
11	final public void getClass()

toString() METHOD

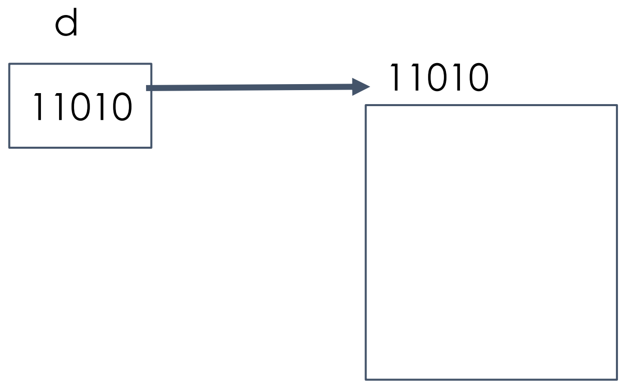
toString()

- toString() method returns String.
- toString() implementation of Object class returns the reference of an object in the String format.

Return Format : ClassName@HexaDecimal

EXAMPLE :

```
class Demo
{
        public static void main(String[] args)
        {
                Demo d = new Demo();
                System.out.println(d) // d.toString() --- Demo@2f92e0f4
        }
}
```



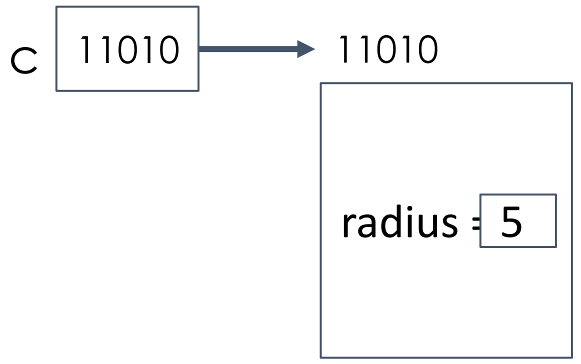
OVERRIDING toString()

PURPOSE OF OVERRIDING toString() :

We override toString() method to print state of an object instead of printing reference of an object.

EXAMPLE :

```
class Circle
{
        Circle(int radius)
        {
                this.radius = radius ;
        }
        Int radius ;
        @Override
        Public String toString()
{
        return "radius : "+radius ;
}
Public static void main(String[] args)
{
        Circle c = new Circle(5);
        System.out.println(c) // c.toString()---radius : 5
}
}
```



NOTE :

- **Java doesn’t provide the real address of an object.**
- **Whenever programmer tries to print the reference variable toString() is implicitly called.**

equals(Object) METHOD

equals(Object) :

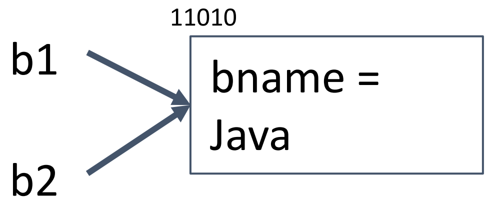
- The return type of equals(Object) method is boolean.
- To equals(Object) method we can pass reference of any object.
- The java.lang.Object class implementation of equals(Object) method is used to compare the reference of two objects.

EXAMPLE 1 :

```
class Book
{
        String bname;
        Book(String bname)
        {
                this.bname = bname;
        }
}
```

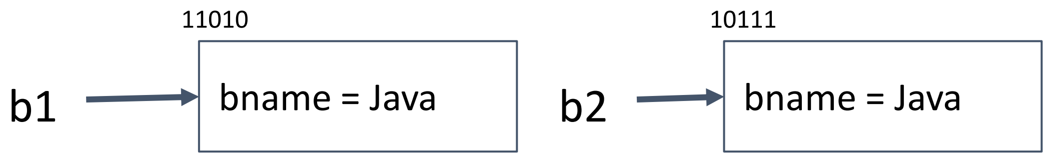
Case 1

```
Book b1 = new Book(“Java”);
Book b2 = b1;
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // true
s.o.pln(b1.equals(b2)); // true
```



Case 2

```
Book b1 = new Book(“Java”);
Book b2 = new Book(“Java”);
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // false
s.o.pln(b1.equals(b2)); // flase
```



NOTE :

- If the reference is same == operator will return true else it returns false.
- The equals(Object) method is similar to == operator.

OVERRIDING equals(Object)

PURPOSE OF OVERRIDING equals(Object) :

We override to equals(Object) method to compare the state of an two Objects instead of comparing reference of two Objects.

NOTE :

- **If equals(Object) method is not overridden it compares the reference of two objects similar to == operator.**
- **If equals(Object) method is overridden it compares the state of two objects, in such case comparing the reference of two objects is possible only by == operator.**

Design tip :

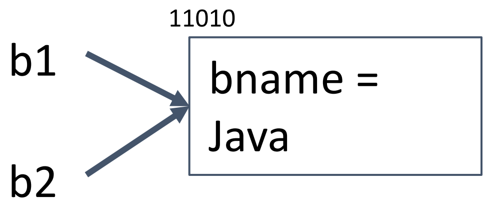
In equals method compare the state of an current(this) object with the passed object by downcasting.

EXAMPLE :

```
class Book
{
        String bname ;
        Book(String bname)
        {
                this.bname = bname ;
        }
        @Override
        public boolean equals(Object o)
        {
                Book b = (Book)o;
                return  this.bname.equals( b.bname) ;
        }
}
```

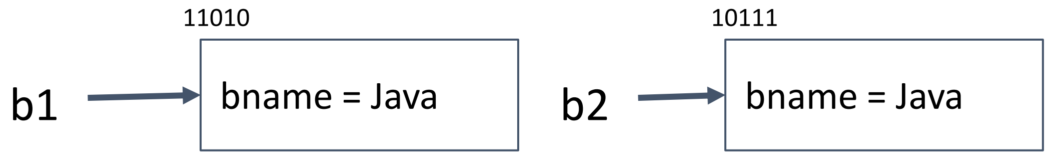
Case 1

```
Book b1 = new Book("Java");
Book b2 = b1;
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // true
s.o.pln(b1.equals(b2)); // true
```



Case 2

```
Book b1 = new Book("Java");
Book b2 = new Book("Java");
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // false
s.o.pln(b1.equals(b2)); // true
```



hashCode() METHOD

hashCode() :

- The return type of hashCode() method is int.
- The java.lang.Object implementation of hashCode() method is used to give the unique integer number for every object created.
- The unique number generated based on the reference of an object.

OVERRIDING hashCode()

PURPOSE OF OVERRIDING hashCode() :

If the equals(Object) method is overridden , then it is necessary to override the hashCode() method.

Design tip :

hashCode() method should return an integer number based on the state of an object.

EXAMPLE 1:

```
class Pen
{
        double price ;
        Pen(double price)
        {
                this.price = price ;
        }
        @Override
        public int hashCode()
        {
                int hc = (int)price;
                return hc ;
        }
}
```

EXAMPLE 2 :

```
class Book
{
        int bid ;
        double price ;
        Book(int bid , double price)
        {
                this.bid = bid ;
                this.price = price;
        }
        @Override
        public int hashCode()
        {
                int hc1 = bid ;
                double hc2 = price ;
                int hc = hc1+(int)hc2;
                return hc ;
        }
}
```

EXAMPLE 1 :

```
class Book
{
        String bname;
        Book(String bname)
        {
                this.bname = bname;
        }
}
```

Case 1

```
Book b1 = new Book("Java");
Book b2 = b1;
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // true
S.o.pln(b1.equals(b2)); // true
int h1 = b1.hashCode();
Int h2 = b2.hashCode();
S.o.pln(h1==h2); // true
```

Case 2

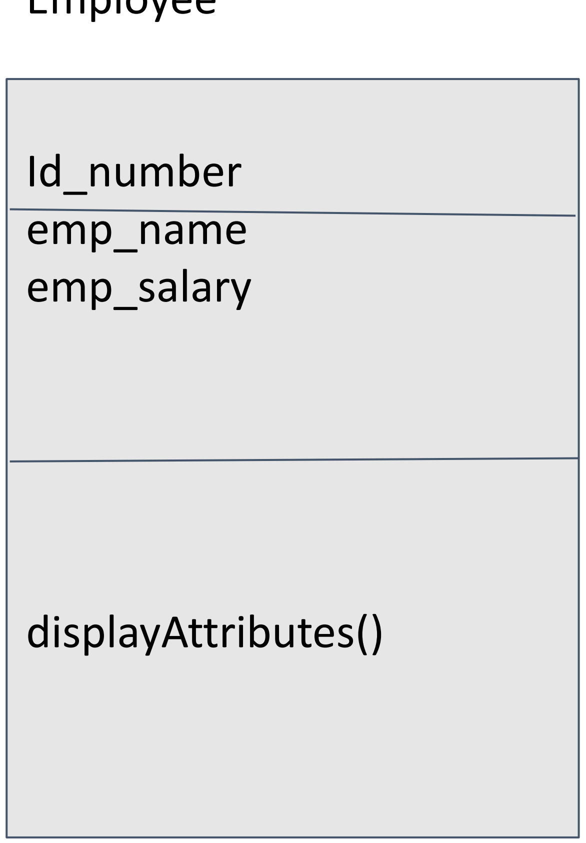
```
Book b1 = new Book("Java");
Book b2 = new Book("Java");
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // false
S.o.pln(b1.equals(b2)); // flase
int h1 = b1.hashCode();
Int h2 = b2.hashCode();
S.o.pln(h1==h2); // false
```

OBSERVATION :

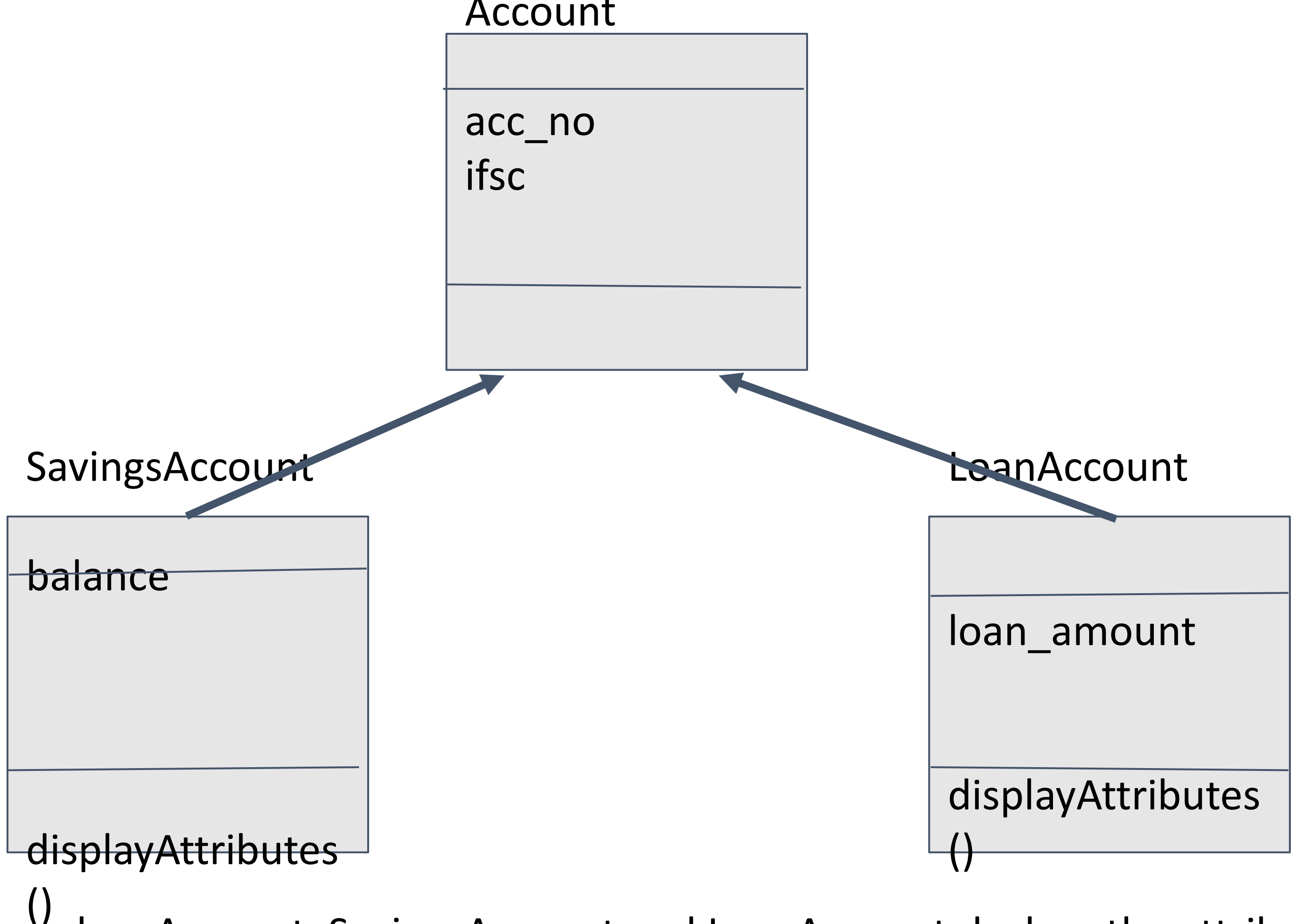
In the above two cases it is clear that,

- If the hashcode for two object is same, equals(Object) method will return true.
- If the hashcode for two object is different, equals(Object) method will return false.

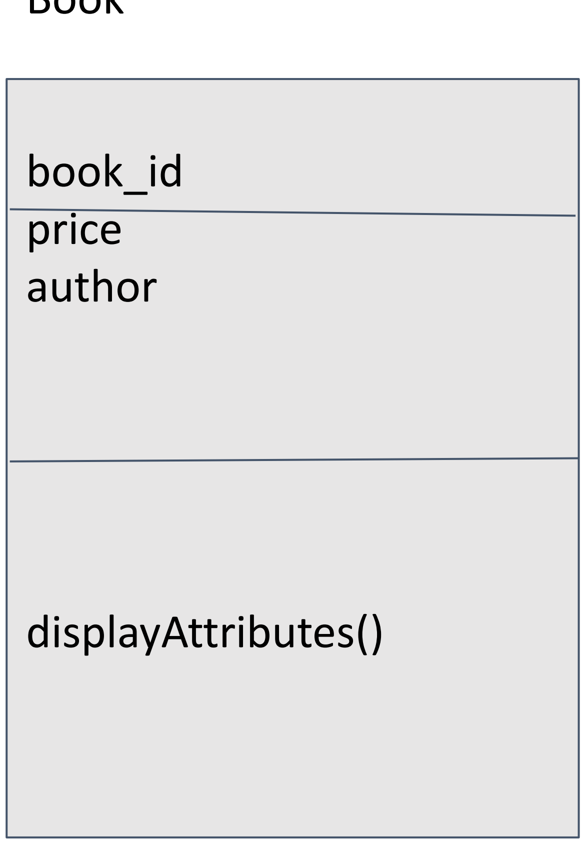
PROGRAMMING SESSION ON OBJECT CLASS



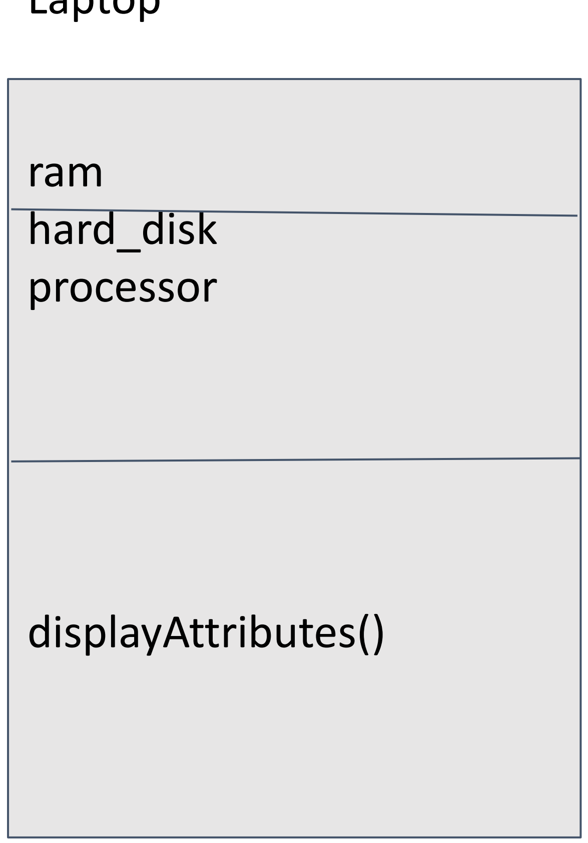
Create a class Employee , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Employee class.
Override the toString(), equals() and hashCode().



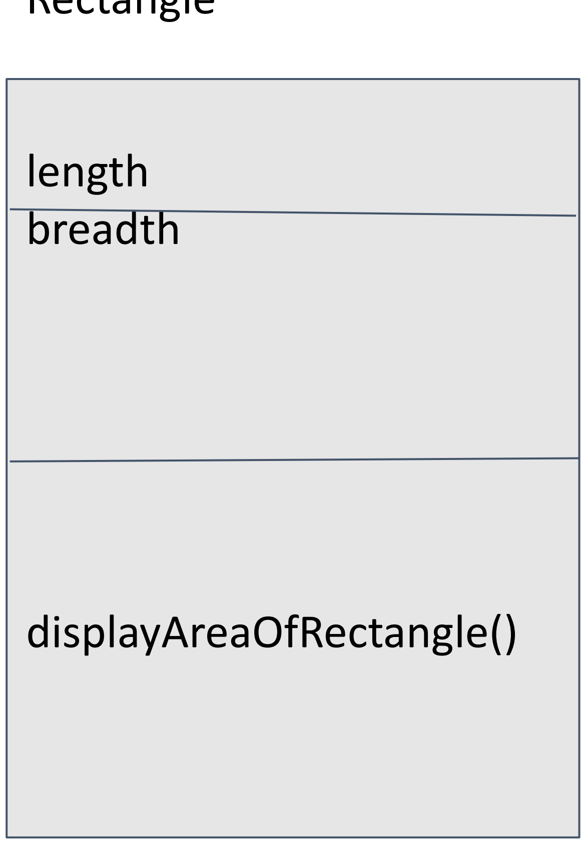
Create a class Account, SavingsAccount and LoanAccount declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for all the class.
Override the toString(), equals() and hashCode().



Create a class Book , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Book class.
Override the toString(), equals() and hashCode().



Create a class Laptop , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Laptop class.
Override the toString(), equals() and hashCode().



Create a class Rectangle , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Rectangle class.
Override the toString(), equals() and hashCode().

STRING

STRING :

String is a literal (data). It is a group of character that is enclosed within the double quote “ ”.

- It is a Non primitive data.
- In java we can store a string by creating instance of the following classes.
    - **java.lang.String**
    - **java.lang.StringBuffer**
    - **java.lang.StringBuilder**
- In java, whenever we create a string compiler implicitly create an instance for java.lang.string in string pool area / string constant pool (scp).

STRING LITERAL

STRING LITERAL:

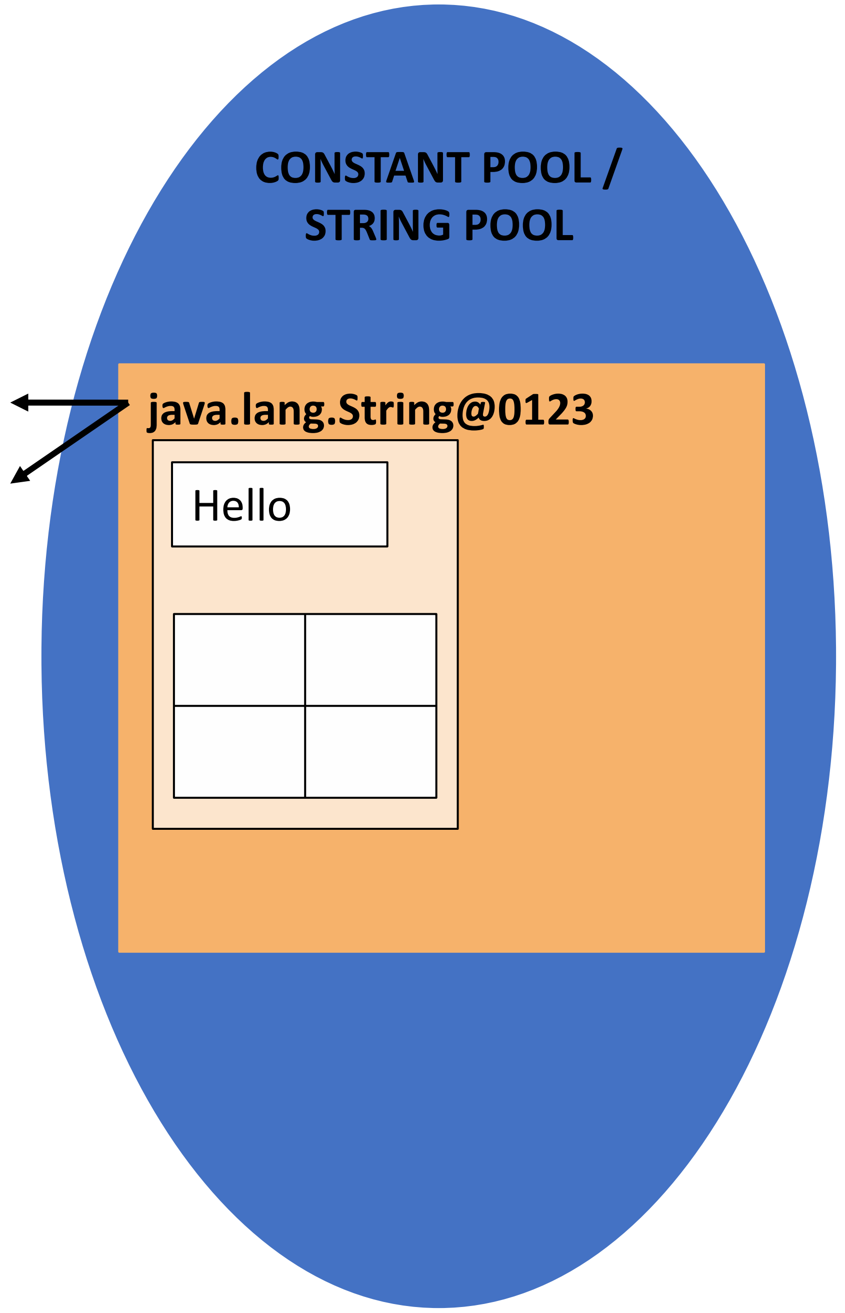
Anything enclosed within the double quote “ ” in java is considered as String literal.

Characteristics of String Literal :

- When a String literals is used in a java program, an instance of java.lang.String class is created inside a String pool.
- For the given String literal, If the instance of a String is already present, then new instance is not created instead the reference of a existing instance is given.

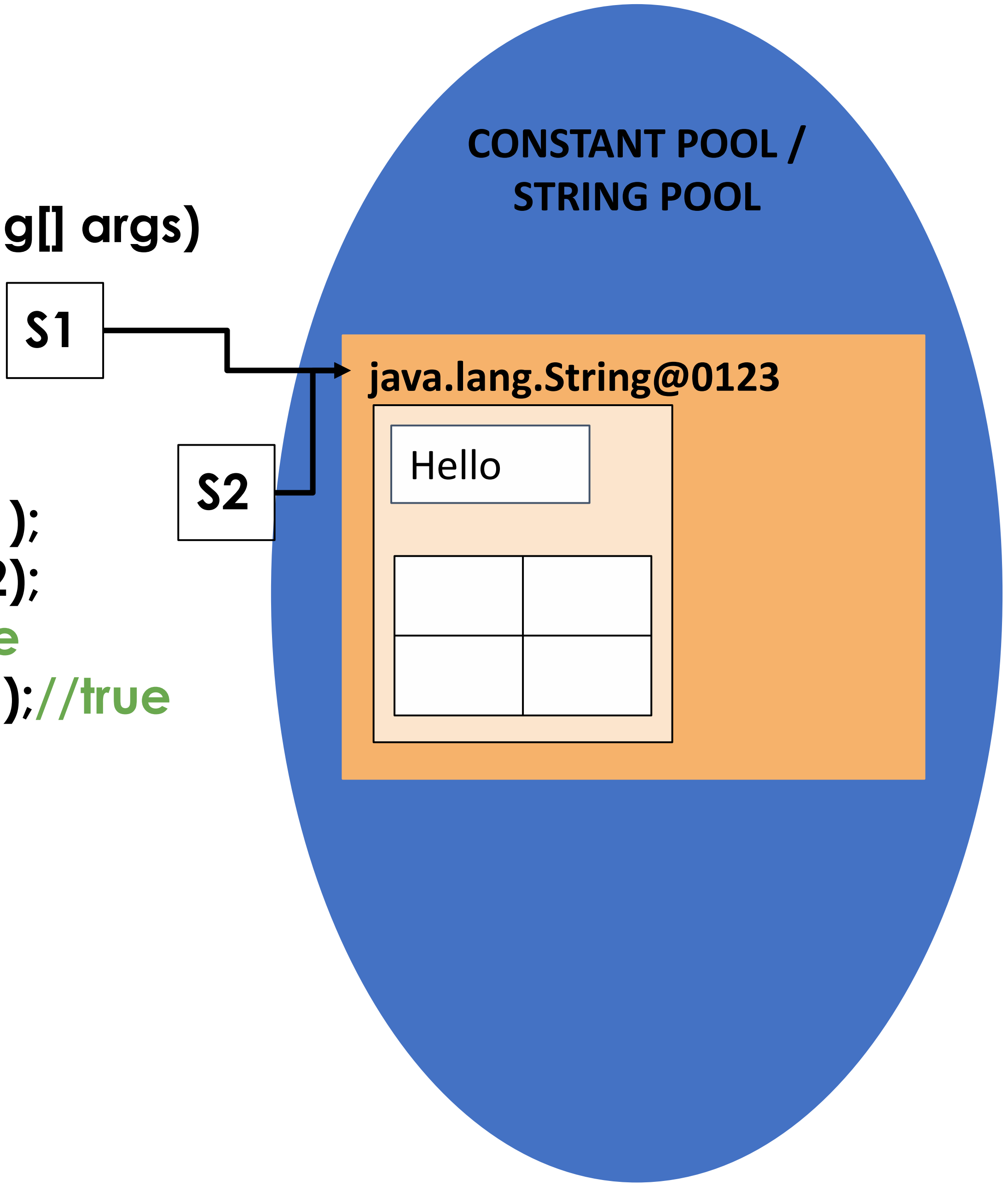
EXAMPLE:1

```
Class Demo
{
        public static void main(String[] args)
        {
                System.out.println("Hello"); //String@0123
                System.out.println("Hello"); //String@0123
        }
}
```



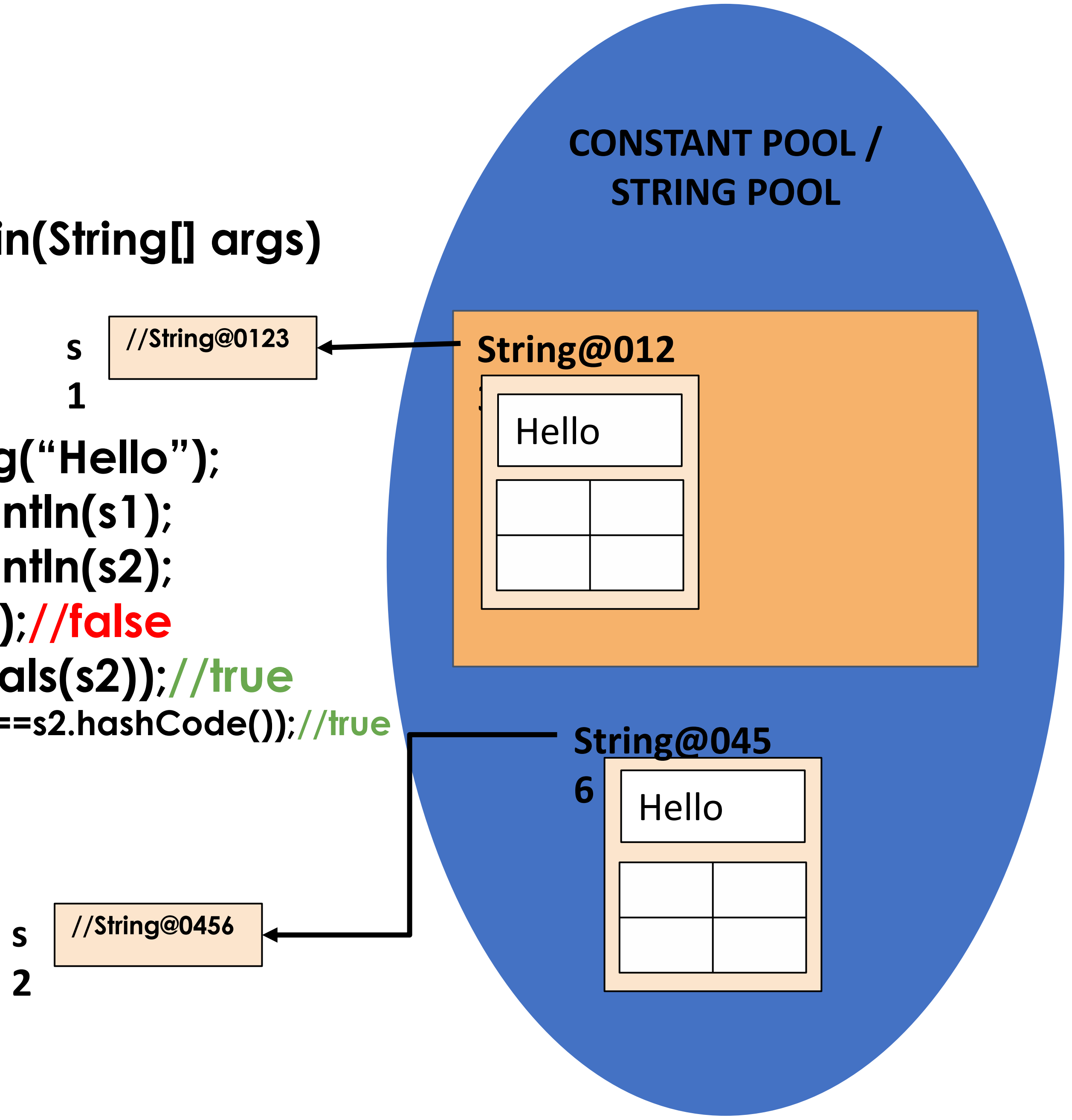
EXAMPLE:2

```
Class Demo
{
        public static void main(String[] args)
        {
                String s1, s2;
                s1 = “Hello”;
                s2 = “Hello”;
                System.out.println(s1);
                System.out.println(s2);
    System.out.println(s1==s2);//true
    System.out.println(s1.equals(s2));//true
}
}
```



EXAMPLE:3

```
Class Demo
{
        public static void main(String[] args)
        {
                String s1, s2;
                s1 = "Hello";
                s2 = new String("Hello");
                System.out.println(s1);
                System.out.println(s2);
    System.out.println(s1==s2);//false
    System.out.println(s1.equals(s2));//true
    System.out.println(s1.hashCode()==s2.hashCode());//true
}
}
```



STRING CLASS

java.lang.String :

- String is a inbuilt class defined in a java.lang package.
- It is a final class.
- It inherits java.lang.Object
- In a String class toString(), equals(), hashCode() methods of java.lang.Object class are overridden.

It implements :

- Comparable
- Serializable
- CharSequence

CONSTRUCTOR IN STRING CLASS

CONSTRUCTORS :

CONSTRUCTORS	DESCRIPTION
String()	Creates an empty string object
String(String literals)	Creates string object by initializing with string literals
String(char[] ch)	Creates String by converting character array into string
String(byte[] b)	Creates String by converting byte array into string
String(StringBuffer sb)	Creates String by converting StringBuffer to string
String(StringBuilder sb)	Creates String by converting StringBuilder to string

METHODS OF STRING CLASS

IMPORTANT METHODS :

RETURN TYPE	METHOD NAME	DESCRIPTION
String	toUpperCase()	Converts the specified string to Upper case
String	toLowerCase()	Converts the specified string to Lowercase
String	concat(String s)	joins the specified Strings
String	trim()	Remove the space present before and after the string
String	substring(int index)	Extract a characters from a string object starts from specified index and ends at the end of a string
String	substring(int start, int end)	Extract a characters from a string starts from specified index and ends at end-1 index

RETURN TYPE	METHOD NAME	DESCRIPTION
char	**charAt(int index)**	Returns character of the specified index from the string
int	**indexOf(char ch)**	Return the index of the character specified if not return -1
int	**indexOf(char ch, int Start_Index)**	Return the index of the character specified by searching from specified index if not return -1
int	**indexOf(charSequence str)**	Return the index of the specified string index if not return -1
int	**indexOf(charSequence str,int Start_Index)**	Return the index of the specified string by searching from specified index if not returns -1
int	**lastIndexOf(char ch)**	Returns the index of the character which is occurred at last in the original String
int	**length()**	Returns length of the string

RETURN TYPE	METHOD NAME	DESCRIPTION
boolean	**equals(Object o)**	Compares states of a two strings
boolean	**equalsIgnoreCase(String s)**	Compares two strings by ignoring its case
boolean	**contains(String str)**	Returns true if specified String is present else it returns false
boolean	**isEmpty()**	Returns true if string is empty else return false
char[]	**toCharArray(String str)**	Converts the specified string into character array
string[]	**split(String str)**	Break the specified string into multiple string and returns String array
byte[]	**getBytes()**	Converts the specified string to byte value and returns byte array

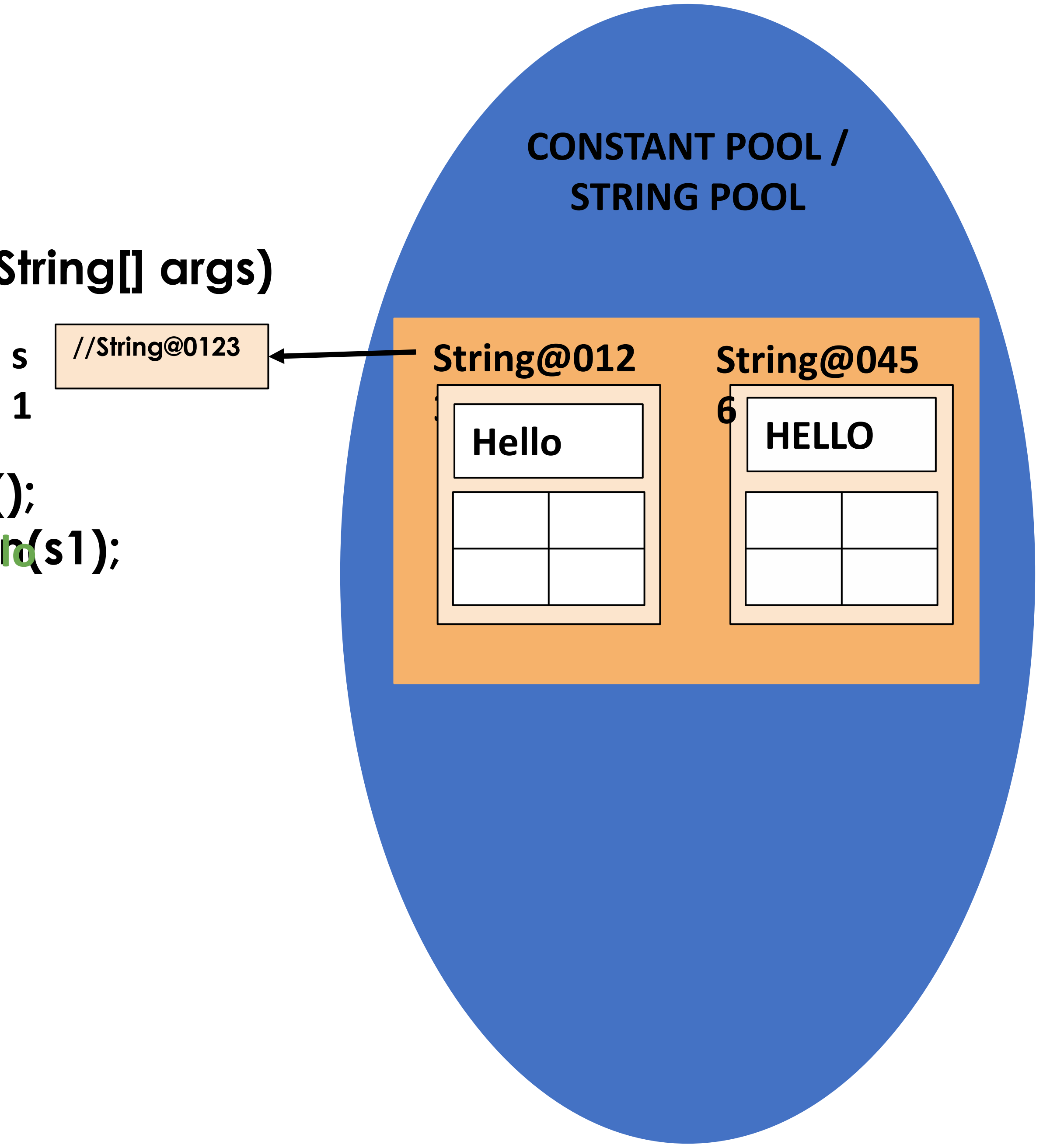
CHARACTERISTICS OF STRING

CHARACTERISTIC :

- Instance of String class is immutable in nature. (Once the object is created then the state is not modified)
- If we try to manipulate (Modify) the state/data then new object is created and reference is given

EXAMPLE:1

```
Class Demo
{
          public static void main(String[] args)
          {
                    String s1, s2;
                    s1 = “Hello”;
                    s1.toUpperCase();
                    System.out.println(s1); // Hello
}
}
```



COMPARISON OF STRING

COMPARISON OF STRING :

- **== ------------------------->**Compares the reference
- **equals() ----------------->**Compares state/data of the object
- **equalsIgnoreCase()--->**Compares the state/data of the object by ignoring its case
- **compareTo()------------>**Compares two string and returns integer value

  Syntax: “String1”.compareTo(“String2”)

  - **string1==string2 ---> 0**
  - **string1>string2-------> +ve Integer**
  - **string1<string2--------> -ve Integer**

DISADVANTAGE OF java.lang.String

DISADVANTAGES :

Immutability, because for every modification separate object is get created in a memory, it reduces the performance.

NOTE :

To overcome the disadvantage of String class we can go for StringBuffer and StringBuilder

EXAMPLE:2 Reversing of a String

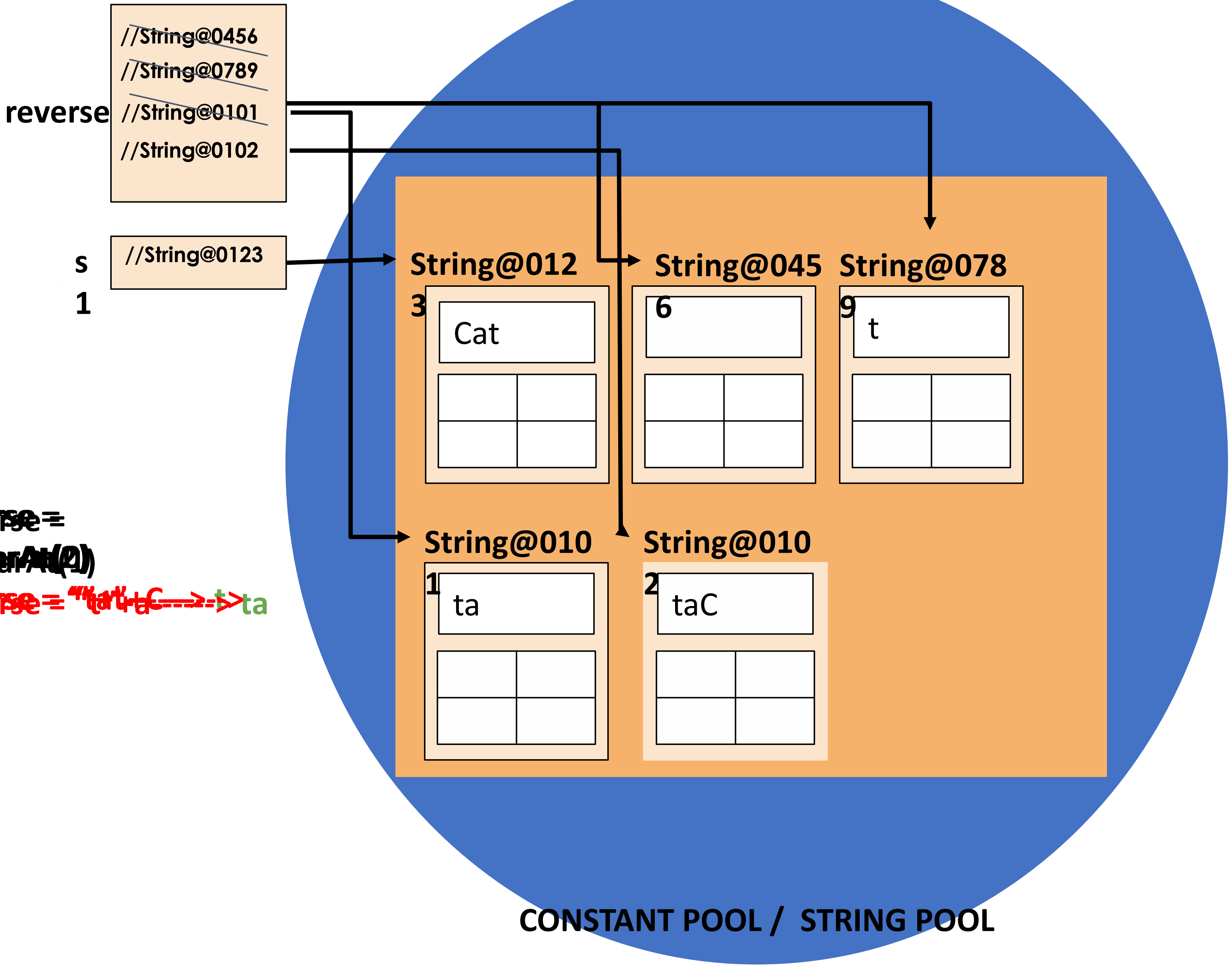
```
Class Demo
{
        public static void main(String[] args)
        {
                String s1=”Cat”;
                String reverse = “”;
                for(int i=s1.length()-1; i>=0; i--)
                {
                        reverse = reverse+s1.charAt(i);
                }
                System.out.println(reverse);
}
}
```

Iteration 2:

reverse =
reverse+S1.charAt(1)
reverse = "t"+a ---> ta

Iteration 3:

reverse =
reverse+S1.charAt(2)
reverse = "ta"+C ---> taC



STRINGBUFFER CLASS

java.lang.StringBuffer :

- It is a inbuilt class defined in java.lang package.
- It is a final class.
- It helps to create mutable instance of String.
- StringBuffer does not have SCP.
- It inherits java.lang.Object class.
- In StringBuffer equals(), hashcode() methods of java.lang.Object class are not overridden.

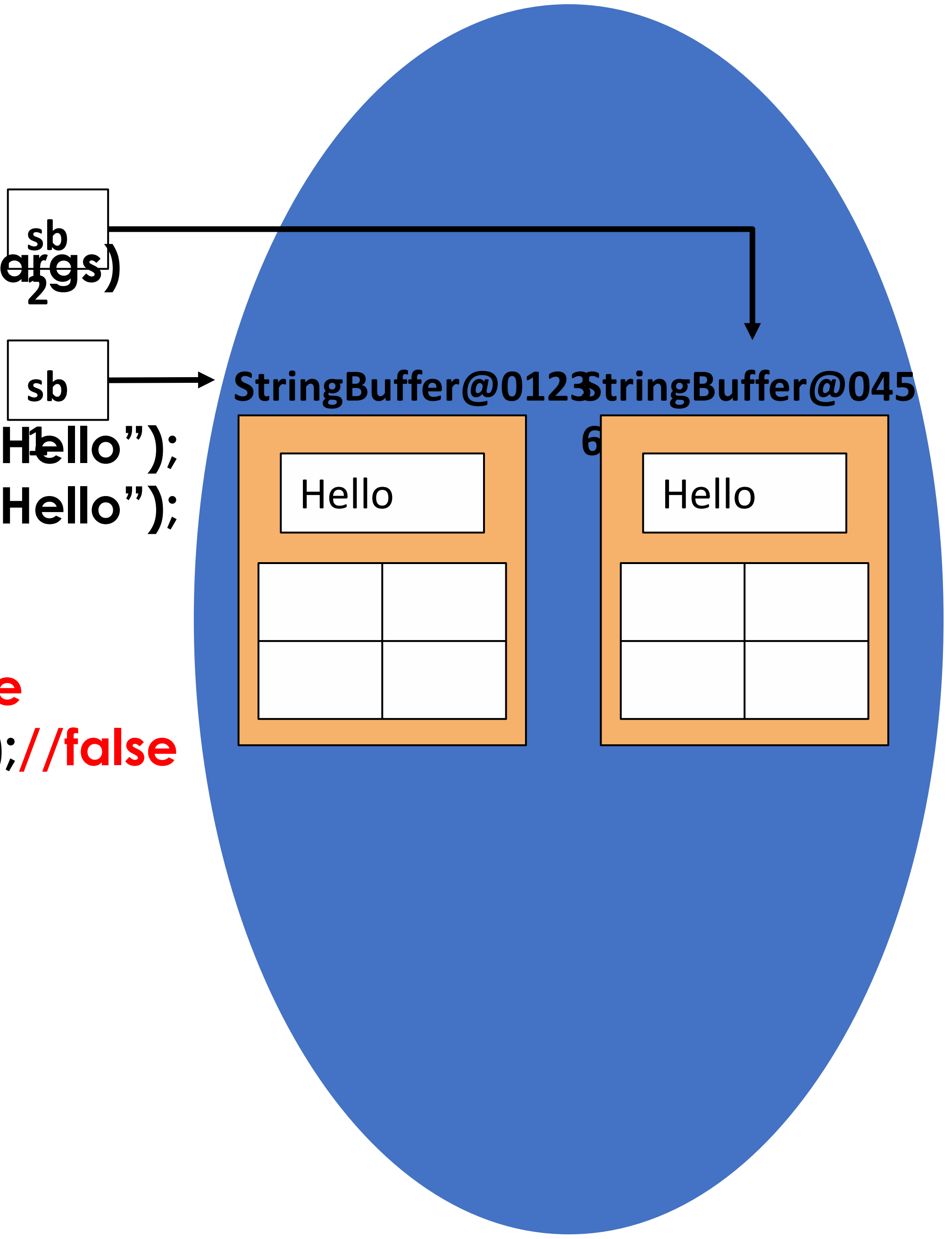
It implements :

Serializable

CharSequence

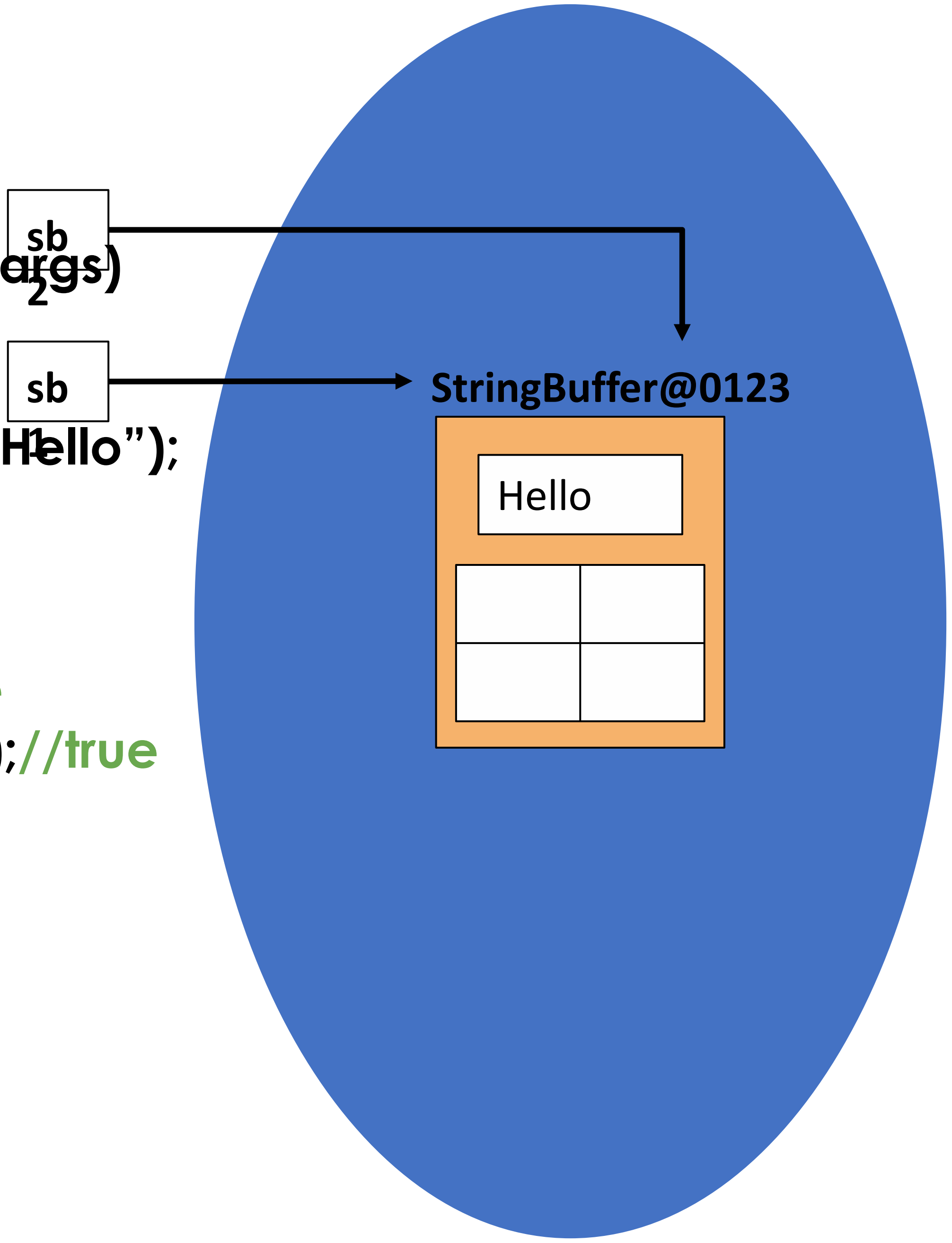
EXAMPLE : 1

```
Class Demo
{
        public static void main(String[] args)
        {
                StringBuffer sb1, sb2;
                sb1 = new StringBuffer("Hello");
                sb2 = new StringBuffer("Hello");
                System.out.println(sb1);
                System.out.println(sb2);
     System.out.println(sb1==sb2);//false
     System.out.println(sb1.equals(sb2));//false
}
}
```



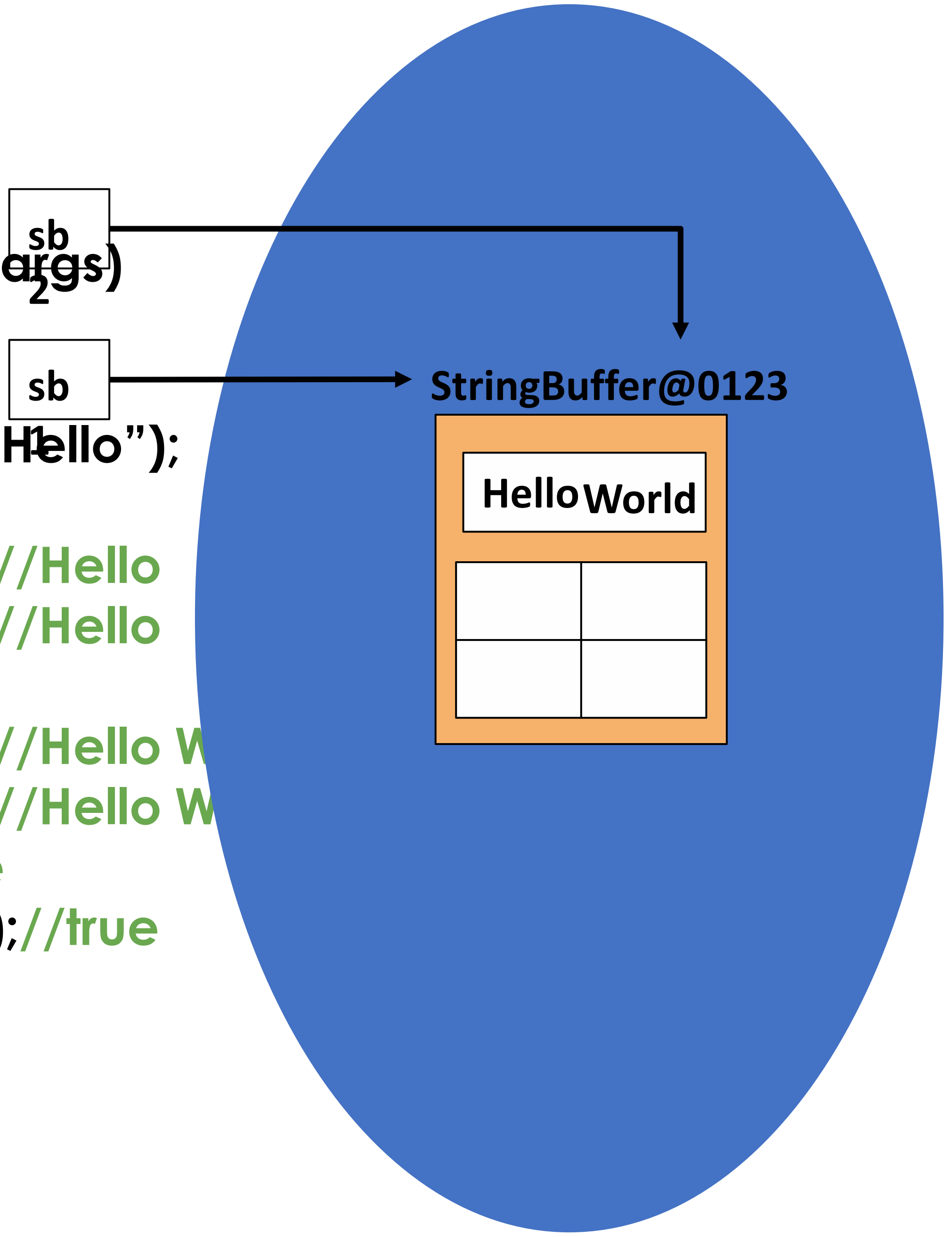
EXAMPLE : 2

```
Class Demo
{
        public static void main(String[] args)
        {
                StringBuffer sb1, sb2;
                sb1 = new StringBuffer("Hello");
                sb2 = sb1;
                System.out.println(sb1);
                System.out.println(sb2);
    System.out.println(sb1==sb2);//true
    System.out.println(sb1.equals(sb2));//true
}
}
```



EXAMPLE : 3

```
Class Demo
{
        public static void main(String[] args)
        {
                StringBuffer sb1, sb2;
                sb1 = new StringBuffer("Hello");
                sb2 = sb1;
                System.out.println(sb1);//Hello
                System.out.println(sb2);//Hello
                sb1.append(" World");
                System.out.println(sb1);//Hello W
                System.out.println(sb2);//Hello W
    System.out.println(sb1==sb2);//true
    System.out.println(sb1.equals(sb2));//true
}
}
```



CONSTRUCTORS

CONSTRUCTORS	DESCRIPTION
StringBuffer()	Creates empty String with initial capacity 16
StringBuffer(String str)	Creates string buffer with the specified string

IMPORTANT METHODS OF STRINGBUFFER

METHODS :

RETURN TYPE	METHOD NAME	DESCRIPTION
int	capacity()	Returns current capacity.
int	length()	Returns length of the string.
char	charAt(int index)	Returns the character of the specified index.
StringBuffer	append(String s)	Join strings. (Overloaded method).
StringBuffer	insert(int index, String s)	Insert a specified string into original String of specified index. (Overloaded method)
StringBuffer	delete(int begin, int end)	Delete String from specified beginning index to end-1 index.

RETURN TYPE	METHOD NAME	DESCRIPTION
StringBuffer	**deleteCharAt(int index)**	Delete the character present in the specified index.
StringBuffer	**reverse()**	Reverse the string
StringBuffer	**setLength(int length)**	Only specified length in a string is exist remaining get removed.
StringBuffer	**substring(int begin)**	Returns the substring from the specified beginning index
StringBuffer	**substring(int begin, int end)**	Returns substring from specified beginning index to end-1 index.
StringBuffer	**replace(int begin,int end,String s)**	Replace a specified string from the beginning index to end-1 index
void	**trimToSize()**	Removes the unused capacity or set the capacity till length of the string
void	**setCharAt(int index, char new char)**	Replace the new character in a string of specified index.
void	**ensureCapacity(int capacity)**	Sets capacity for storing a string.

CHARACTERISTICS OF STRINGBUFFER

CHARACTERISTICS :

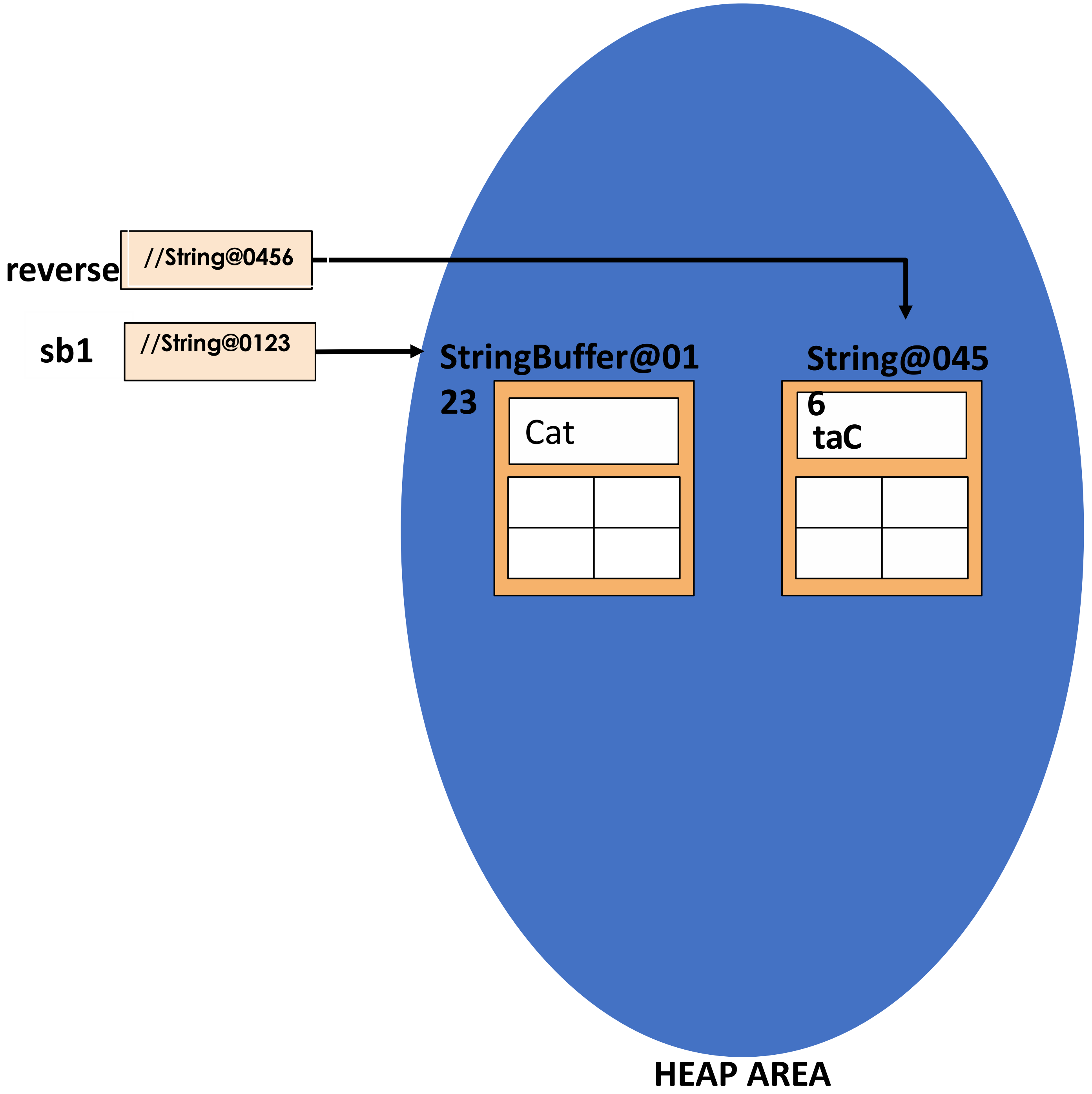
It is immutable.

NOTE :

String constant pool is not applicable to String Buffer.

EXAMPLE :

```
class StringBufferReverseDemo
{
        public static void main(String[] args)
        {
                StringBuffer sb1=new StringBuffer("Priyatharsan");
                StringBuffer reverse = new StringBuffer();
                for(int i=sb1.length()-1; i>=0 ; i--)
                {
                        reverse.append(sb1.charAt(i));
                }
                System.out.println(reverse);
        }}
```



Iteration 3:

reverse.append(S1.charAt(

(0))--taC

COMPARISON OF STRINGBUFFER

COMPARISON OF STRINGBUFFER :

- **== ------------------------->**Compares the reference.
- **equals() ----------------->**Compares reference of the object.

DISADVANTAGE OF STRINGBUFFER

DISADVANTAGES :

Multiple thread can't execute the StringBuffer object simultaneously because all the methods are synchronized. So, Execution time is more. In order to overcome this problem we will go for String Builder.

NOTE :

The characteristics of StringBuffer and StringBuilder are same

DIFFERENCE BETWEEN STRINGBUFFER AND STRINGBUILDER

STRING BUFFER	STRING BUILDER
All the method present in StringBuffer is synchronized.	All the method present in StringBuilder is non synchronized.
At a time only one thread is allowed to access String Buffer object. Hence it is Thread safe.	At a time multiple thread is allowed to access String Builder object. Hence it is Not Thread safe.
Threads are required to wait to operate a stringBuffer object. Hence Relatively performance is low .	Threads are not required to wait to operate a StringBuilder object. Hence Relatively performance is high.
Less efficient than StringBuilder	Efficiency is high compared to StringBuffer
Introduced in 1.0 v	Introduced in 1.5v

ABSTRACTION

ABSTRACTION :

It is a design process of hiding the implementation and showing only the functionality (only declaration) to the user is known as abstraction.

HOW TO ACHIEVE ABSTRACTION IN JAVA ?

- In java we can achieve abstraction with the help of abstract classes and interfaces.
- We can provide implementation to the abstract component with the help of inheritance and method overriding.

ABSTRACT MODIFIER

ABSTRACT MODIFIER :

- Abstract is a modifier, it is a keyword.
- It is applicable for methods and classes.

ABSTRACT METHOD :

- A method that is prefixed with an abstract modifier is known as the abstract method.
- This is also said to be an incomplete method.
- The abstract method doesn’t have a body (it has the only declaration)

Syntax to create abstract method :

```
        abstract[access modifier] [modifier] returnType methodName([For_Arg]) ;
```

NOTE :

Only child class of that class is responsible for giving implementation to the abstract method.

ABSTRACT CLASS

ABSTRACT CLASS :

If the class is prefixed with an abstract modifier then it is known as abstract class.

We can't create the object (INSTANCE) for an abstract class.

NOTE :

- **We can't instantiate an abstract class**
- **We can have an abstract class without an abstract method.**
- **An abstract class can have both abstract and concrete method.**
- **If a class has at least one abstract method either declared or inherited but not overridden it is mandatory to make that class as abstract class.**

EXAMPLE :

```
abstract class Atm
{
        abstract public double withdrawal();
        abstract public void getBalance();
        abstract public void deposit();
}
```

// hiding implementation by providing only functionality

Atm a = new Atm() // CTE

NOTE :

Only subclass of Atm is responsible for giving implementation to the methods declared in a Atm class.

CONCRETE CLASS

CONCRETE CLASS :

- If the class which is not prefixed with an abstract modifier then it is known as concrete class.
- In java, we can create an object (INSTANCE) only for concrete class.

IMPLEMENTATION OF ABSTRACT METHOD

Implementation of abstract method :

- If a class extend abstract class then it should give implementation to all the abstract method of the superclass.
- If inheriting class doesn’t like to give implementation to the abstract method of superclass then it is mandatory to make subclass as an abstract class.
- If a subclass is also becoming an abstract class then the next level child class is responsible to give implementation to the abstract methods.

STEPS TO IMPLEMENT ABSTRACT METHOD :

STEP 1:

Create a class.

STEP 2:

Inherit the abstract class/ component.

STEP 3:

Override the abstract method inherited (Provide implementation to the inherited abstract method).

EXAMPLE :

```
abstract class WhatsApp
{
        abstract public void send();
}
class Application extends WhatsApp
{
        public void send()
        {
                System.out.println("Send() method is implemented");
        }
}
```

Creating object and calling the abstract method :

```
WhatsApp w = new Application();
w.send() // Send() method is implemented
```

CONCRETE CLASS :

The class which is not prefixed with an abstract modifier and doesn't have any abstract method, either declared or inherited is known as concrete class.

NOTE :

In java we can create object only for the concrete class.

CONCRETE METHOD :

The method which give implementation to the abstract method is known as concrete method.

INTERFACES

INTERFACES :

It ia a component in java which is used to achieve 100 % abstraction with multiple implementation.

Syntax to create an interface

```
[Access Modifier] interface InterfaceName
{
    // declare members
}
```

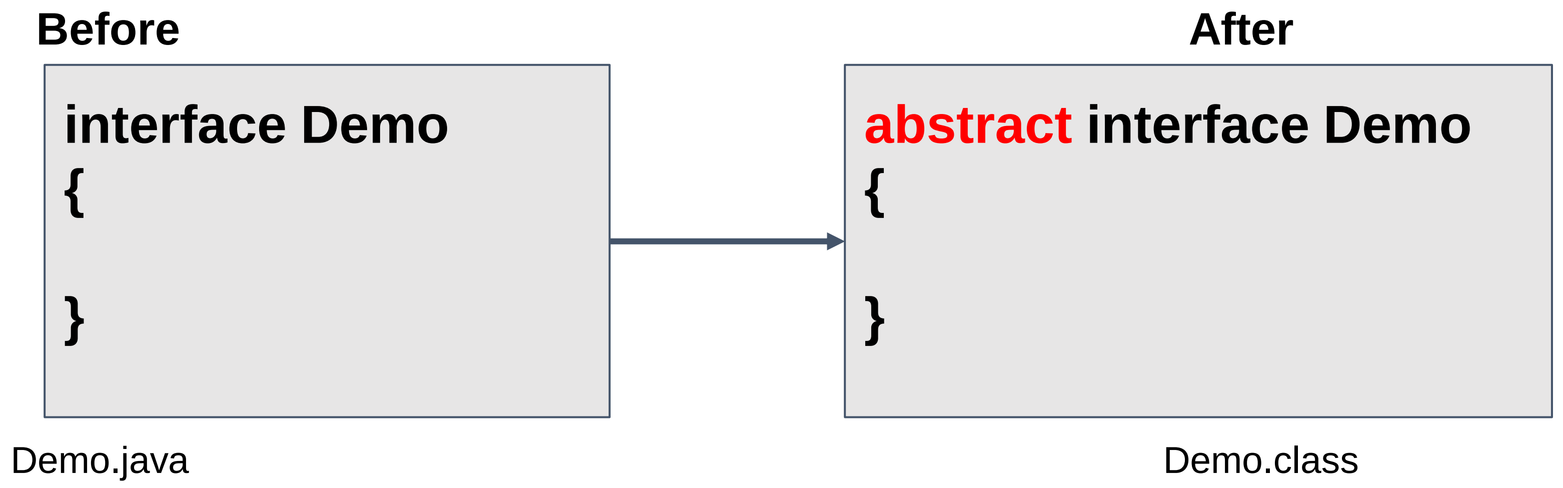
When an interface is compiled we get a class file with extension.class only

EXAMPLE :

```
interface Demo
{

}
```

Demo is an interface



```
interface Demo
{

}
```

```
abstract interface Demo
{

}
```

```
Interface Demo1
{
            Int a; //CTE : Variable a is by default public, static, final. A final variable must be initialised.

}
```

```
 Interface Demo2
{
            Public void test()// CTE
            {

      }
}
```

```
Interface Demo3
{
            Static void main(String[] a)
            {
                        System.out.println("Hello World..!!!");
      }
}
```

What all are the members the can be declared in an interface ?

MEMBERS	CLASS	INTERFACE
Static variables	Yes	Yes, but only final static variables
Non static variables	Yes	No
Static methods	Yes	Yes, From JDK 1.8 v. NOTE : They are by default public in nature
Non static methods	Yes	Yes but we can have only abstract non static methods NOTE : Non static methods are by default ● public ● abstract
Constructors	Yes	No
Initializers (Static & non static block)	Yes	No

NOTE :

In interface all the members are by default have public access modifier

Why do we need an interface ?

- To achieve 100% abstraction.Concrete non static methods are not allowed.
- To achieve multiple inheritance.

What all the members are not inherited from an interface ?

Only static methods of an interface is not inherited to both class and Interface.

INHERITANCE

INHERITANCE WITH RESPECT TO INTERFACE :

An Interface can inherit any number of interfaces with the help of extend keyword.

EXAMPLE :

```
interface I1
{
}
interface I2 extends I1
{
}
```

NOTE :

The interface which is inheriting an interface should not give implementation to the abstract methods.

EXAMPLE 1 :

- Interface I1 have 3 methods

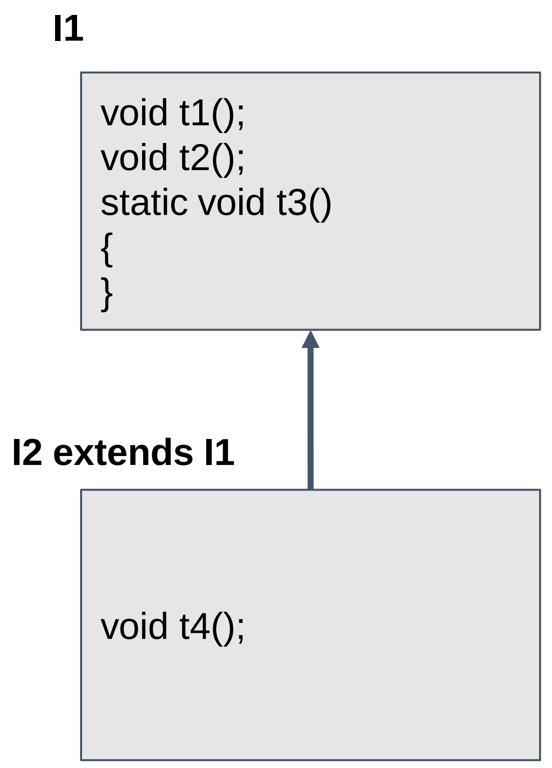
  2 - non static (**t1() , t2()**)

  1 - static (**t3()**)

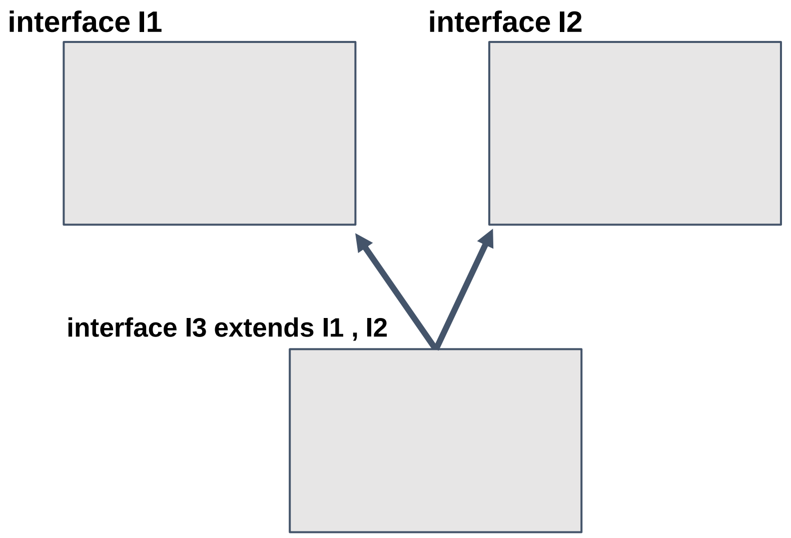
- Interface I2 have 3 methods

  2 - inherited non static method (**t1() , t2()**)

  1 - declared non static methods (**t4()**)



EXAMPLE 2 : Interface can inherit multiple interfaces at a time



NOTE :

With respect to interface there is no diamond problem.

Reason ,

- They don't have a constructors.
- Non static methods are abstract (do not have implementation)
- Static methods are not inherited.

INHERITANCE OF AN INTERFACE BY THE CLASS :

- Class can inherit an interface with the help of implements keyword.
- Class can inherit more than one interface.
- Class can inherit a class and an interface at a time.

NOTE :

- **If a class inherit an interface then it should give implementation to the abstract non static methods of an interface.**
- **If the class is not ready to give implementation to the abstract non static methods of an interface then it is mandatory to make that class as abstract method.**
- **Next level of child class is responsible for giving implementation to the rest of abstract non static methods of an interface.**

EXAMPLE 1 : Class inheriting an interface

- Interface I1 have 3 methods

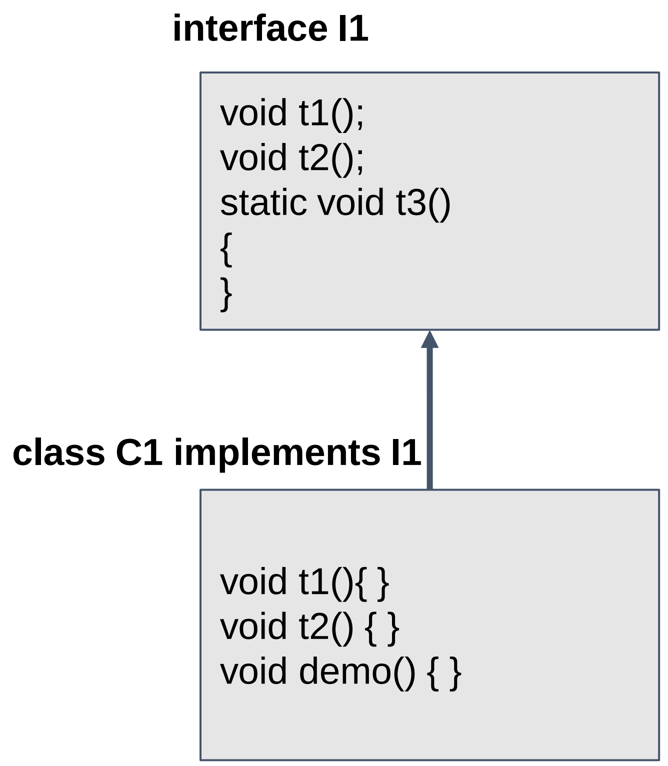
  2 - non static (**t1() , t2()**)

  1 - static (**t3()**)

- Class c1 have 3 methods

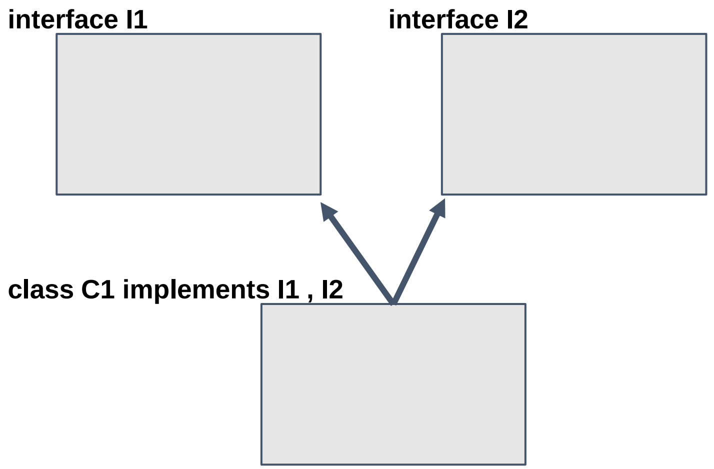
  2 - inherited and implemented non static method (**t1() , t2()**)

  1 - declared non static methods (**demo()**)



SOLUTION FOR DIAMOND PROBLEM

EXAMPLE 2 : Class inheriting multiple interfaces

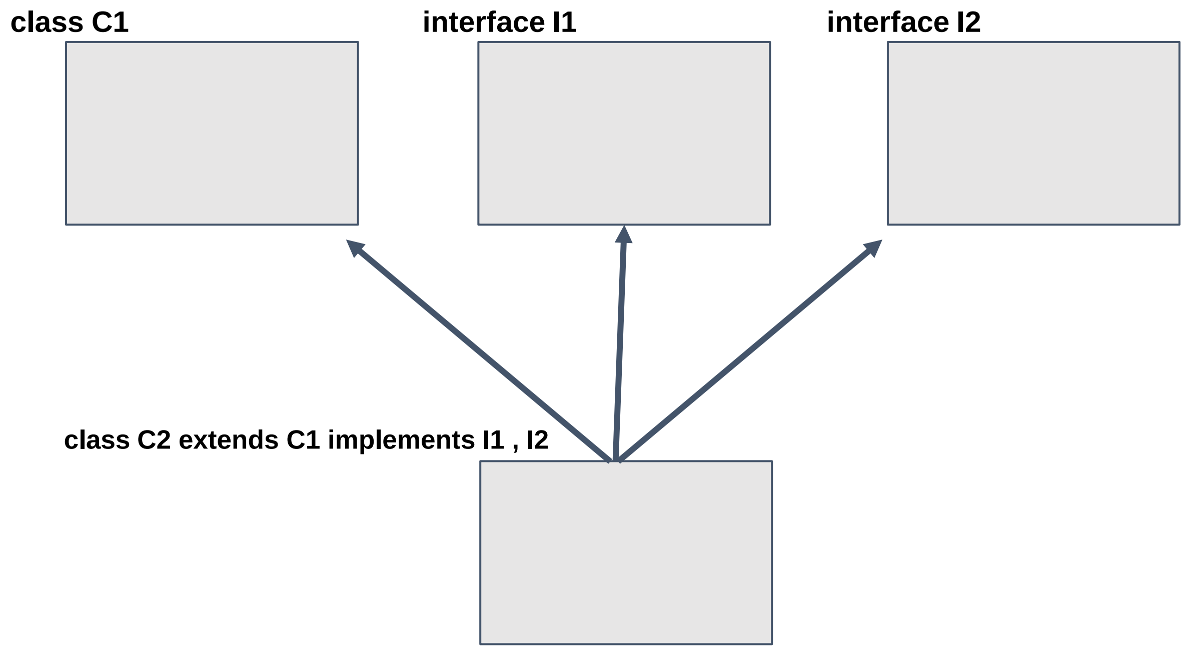


The diamond problem is solved by implementing multiple interface by the class at a time.

Reason,

- Non static methods are not implemented in an interface.
- Static methods are not inherited.

EXAMPLE 3 : Class can inherit interface and class at a time



RULE :

Use the extends first and then implements.

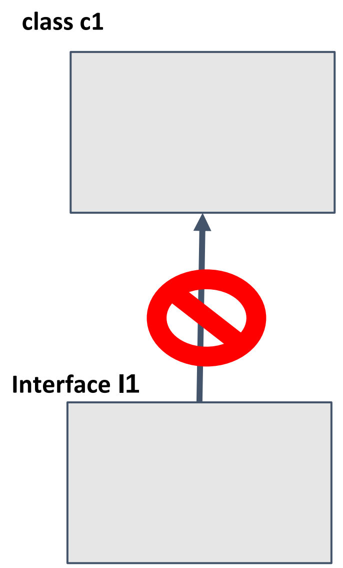
NOTE :

Class can inherit multiple interface but it can’t inherit multiple class at a time.

Interface can’t inherit a class.

NOTE

Interface can't inherit from the class



Reason,

Class has concrete non static methods, So class can't be a parent to the interface

PACKAGES

PACKAGES :

A package in java is used to group a related classes , interfaces and subclasses. In a simple word it is a folder / directory which consist of several classes and interfaces.

NOTE :

Package contains only class file

WHY PACKAGE ? :

- Packages are used to avoid name conflict.
- It increases maintainability.
- It is used to categorize classes and interfaces.
- It increases the access protection.
- It is used to achieve code reusability.

Types of packages :

We have two types of packages

- **Built in packages**
- **User defined packages**

BUILT IN PACKAGES

BUILT IN PACKAGES :

In java we have many built in packages like java, lang, util, io, sql, swing, awt, etc…Which are included in Java Development Kit.

EXAMPLE :

Java.lang.Math

java ---> package

lang ---> subpackage

Math ---> class

Subpackage :

Package inside a package is called as sub package.it should created to categorize a folder further.

HOW TO USE BUILT IN PACKAGES ?

We can use inbuilt packages by using two ways,

1. **By using fully qualified name.**
2. **By using import keyword.**

1.By using fully qualified name :

By using fully qualified name, compiler can understand to which package the specified class is available.

EXAMPLE :

There is one class named as ArrayList is available in java.util package. This class is used to group heterogeneous object as single entity. The fully qualified name for this class is java.util.ArrayList.

Creating object for ArrayList by using fully qualified name,

```
java.util.ArrayList al = new java.util.ArrayList();
```

By using fully qualified name, compiler can understand that ArrayList belongs to java.util Package.

DISADVANTAGE OF USING FULLY QUALIFIED NAME:

- We need use fully qualified name for every time when we accessing the class or interface.
- Readability is low.

To overcome this we can use a class by using import statement,

BY USING IMPORT STATEMENT :

- Import statement is used to import the classes or interfaces present in packages / Subpackages.

Syntax to use import statement:

import package.subpackage/class/interface ;

- By using import statement, instead of using fully qualified name for the classes we can directly use the class name.

RULE TO USE IMPORT STATEMENT:

1. **Import statement should be used before declaring a class.**
2. **import statement should be end with (;).**
3. **We can use multiple import statement in a same program.**

EXAMPLE:

```
import java.util.Scanner ;
class Demo
{
    Public static void main(String []args)
    {
            Scanner s = new Scanner(System.in);
    }
}
```

- By using this statement only Scanner class present inside the java.util package is get imported and accessible.
- If you want to use another class from java.util package use the import statement again before the class.

ADVANTAGE OF IMPORT STATEMENT :

- We can directly use class name instead of using fully qualified.
- By importing a package once we can use the class / interface of that package as many times possible.

NOTE :

java.lang package is implicitly imported by the compiler.

USER DEFINED PACKAGES

USER-DEFINED PACKAGE :

In java we can create our own package.

Syntax to create a package :

```
package package_Name;
```

Subpackage :

We can create subpackage by using following syntax.

Syntax to create package along with subpackage subpackage :

```
package package_name.subpackage_name ;
```

RULE :

- Package should be the first statement in a java program.
- A java source file should contain only one package Statement.
- A package can contain multiple classes/Interfaces but utmost one class/interface should be public.
- If you want to add more then one public class inside the same package then, create a separate source file for each public class with same package name.
- If a package contains public class/interface then it is mandatory to use public class/interface name as a source file name. Otherwise, we will get **Compile Time Error.**

EXAMPLE :

```
package mypack;
public class PackageDemo
{
public static void main(String[] args)
{
    System.out.println("Package is created successfully");
}
}
```

To compile : javac -d . PackageDemo.java

To execute : java mypack PackageDemo

NOTE :

If we want to use package and import statement in a same program then we should follow a sequence of program,

> **package**
>
> **import**
>
> **class / interface / enum**

ACCESS MODIFIERS

We have two type of modifiers

1. Access modifiers
2. Non access modifiers

Access modifiers :

- Access modifier are responsible to change / modify the accessibility of the member.
- We have four type of access modifiers,

    1. **private**
    2. **default**
    3. **protected**
    4. **Public**

PRIVATE

PRIVATE ACCESS MODIFIER :

- It is a class level modifier , it is applicable for variables , method and constructors.
- If the member of a class is prefixed with private modifier then it is accessible only within the class accessing outside the class is not possible.

EXAMPLE :

```
class A{
private static int i ;
}
class B{
Public static void main(String[] args){
System.out.println(A.i); // CTE
}
```

DEFAULT

DEFAULT ACCESS MODIFIER :

- The accessibility of default modifier is only within the package.It can’t be accessed from outside the package.
- If you don’t declare any access modifier then it is considered as a default access modifier.

EXAMPLE :

```
package myPack ;

public class Demo{

            static int i ;

}

class Driver{

Public static void main(String[] args){

Demo.i = 5 ; // CTE

}
```

PROTECTED ACCESS MODIFIER :

- The access level of a protected modifier is within the package and outside the package through child class.
- If you do not make the child class, it cannot be accessed from outside the package.

PUBLIC ACCESS MODIFIER :

- The access level of a public modifier is anywhere.
- It can be accessed from within the class, outside the class, within the package as well as outside the package.

SCOPE OF AN ACCESS MODIFIER

ACCESS MODIFIER	WITHIN THE CLASS	WITHIN THE PACKAGE	OUTSIDE THE PACKAGE	OUTSIDE THE PACKAGE BY THE CHILD CLASS
private	YES	NO	NO	NO
default	YES	YES	NO	NO
protected	YES	YES	YES	NO
public	YES	YES	YES	YES

The scope of the access modifier based on the accessibility is :

private < default < protected < public

ARRAY

ARRAY :

Array is a continuous block of memory which is used to store multiple values.

CHARACTERISTIC OF AN ARRAY :

- The size of an array must be defined at the time of declaration.
- Once declared , the size of an array can’t modified.
- Hence array is known as fixed size.
- In an array we can access the elements with the help of an index or subscript. It is an integer number that starts from 0 and ends at length of the array-1.
- In an array we can store only homogeneous type value. It is also known as homogeneous collection of an object.

NOTE :

In java array is an object

DECLARING AN ARRAY

Syntax to declare an array :

datatype[] variable; (or) datatype variable[]

EXAMPLE :

int a[] **---** single dimensional array reference variable of int type

float f[] **---** single dimensional array reference variable of float type.

String s[] **---** single dimensional array reference variable of String type.

INSTANTIATING AN ARRAY

Syntax to instantiate an array :

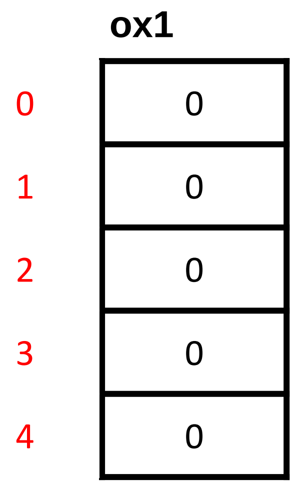
new datatype[size];

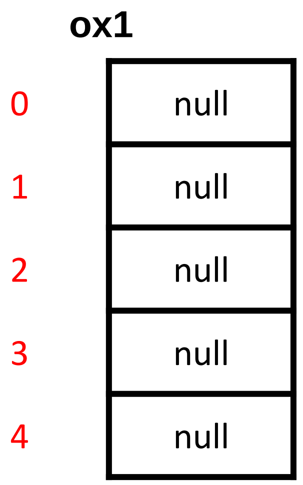
EXAMPLE :

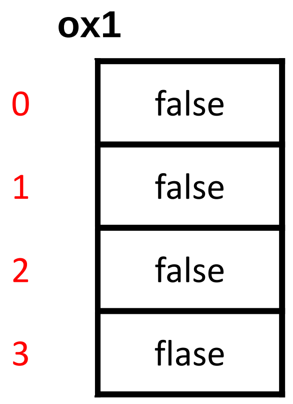
new int[5];

new boolean[4]

new String[5];







NOTE :

Once the array is instantiated it is assigned with default value

INITIALIZING AN ARRAY

ADDING ELEMENTS :

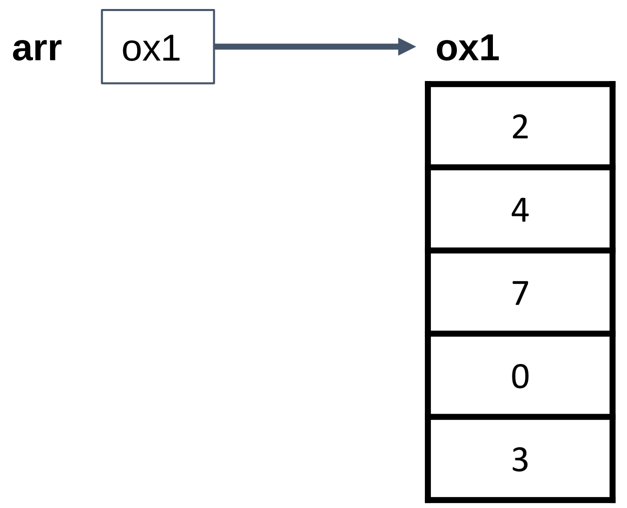
We can add an element into an array with the help of array index.

Syntax to add an element into an array :

array_ref_variable[index] = value ;

EXAMPLE :

```
int[] arr = new int[5] ; //declaration
arr[0] = 2;
arr[1] = 4;
arr[2] = 7;
arr[3] = 0;
arr[4] = 3;
```



ACCESSING ELEMENTS FROM AN ARRAY

ACCESSING ELEMENTS :

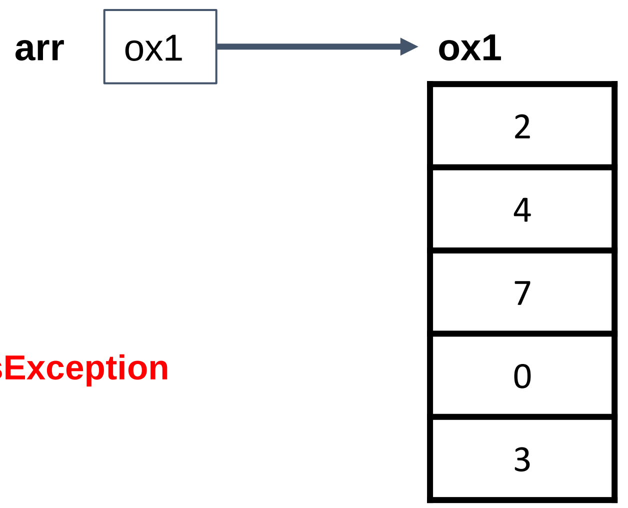
We can access an element from an array with the help of array reference variable and index.

Syntax to access an elements from an array :

array_ref_variable[index] ;

EXAMPLE :

```
System.out.println(arr[0]); // 2
System.out.println(arr[2]); // 7
System.out.println(arr[4]); // 3
System.out.println(arr[5]); //ArrayIndexOutOfBoundsException
```



Declaring , Instantiating , Initializing An Array In A Single Line

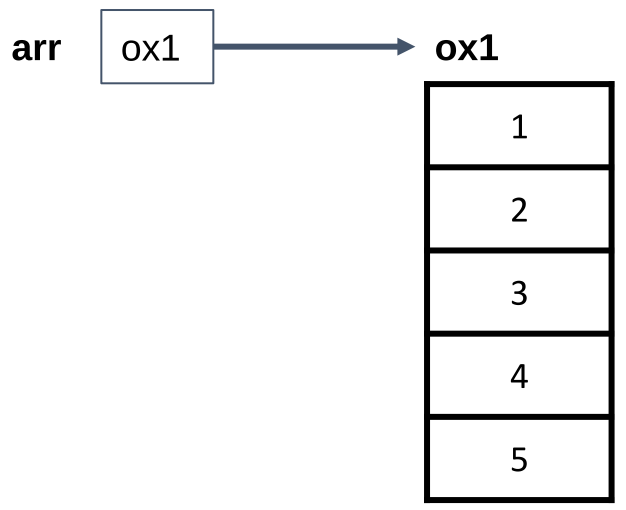
We can also declare , instantiate and initialize an array by using single line.

Syntax :

datatype[] arr_ref_var = { element1 , element2 , element3 , etc... } ;

EXAMPLE :

int arr[] = { 1 , 2 , 3 , 4 , 5 };



EXCEPTION

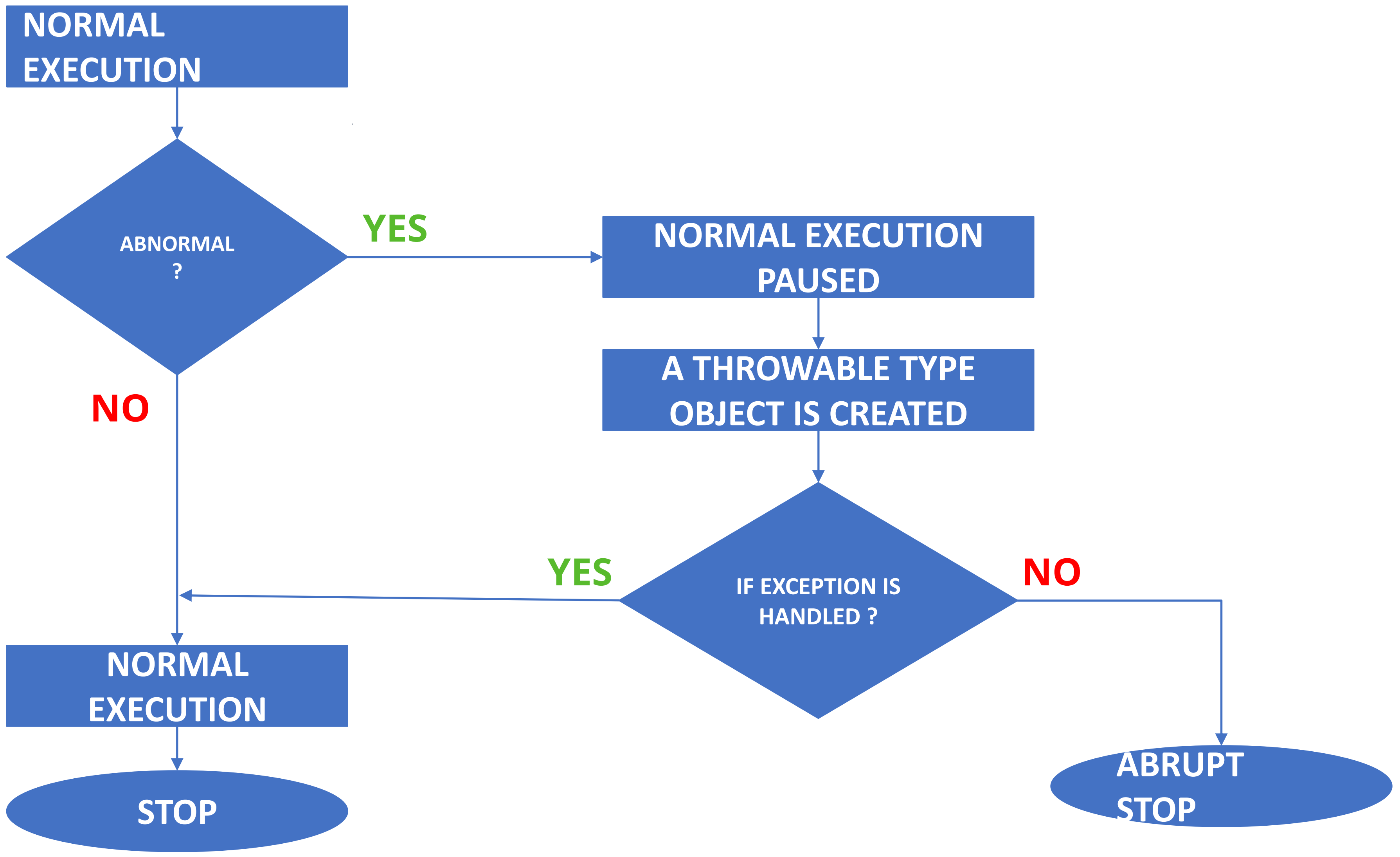
EXCEPTION :

The exception is a problem that occurs during the execution of a program (Runtime). When an exception occurs, the execution of the program stops abruptly (Unexpected stop).

NOTE :

Every exception in java is a class of 'Throwable type'

WHAT HAPPENS IF AN EXCEPTION OCCURS ?



NOTE :

- Every exception is occurred because of a statement.
- A statement will throw an exception during the abnormal situation.

EXAMPLE :

```
        import java.util.Scanner ;
    class
        {
                public static void main(String[] args)
                {
                        Scanner input = new Scanner(System.in);
                        int a , b ;
                        a = input.nextInt();
                        b = input.nextInt();
                        int c = a / b ; // Statement might cause
Exception(ArithmeticException)
                        System.out.println("The division of "+a+" and "+b+" = "+c);
                }
        }
```

CASE 1	CASE 2
a = 5 , b = 10 ; **->**Normal situation **->**No exception	**a = 5 , b = 0 ;** **->**Abnormal situation **->**Exception occurs

IMPORTANT EXCEPTIONS AND STATEMENTS

STATEMENT	EXCEPTION
a/b	ArithmeticException
reference.member	NullPointerException
(ClassName)reference	ClassCastException
array_ref[index]	ArrayIndexOutOfBoundsException
string.charAt(index)	StringIndexOutOfBoundsException
string.substring(Index)	StringIndexOutOfBoundsException

CHECKED EXCEPTION

CHECKED EXCEPTION :

The compiler-aware exception is known as the checked exception. i.e., the Compiler knows the statement responsible for abnormal situations (Exception). Therefore the compiler forces the programmer to either handle or declare the exception. If it is not done we will get an unreported compile-time error.

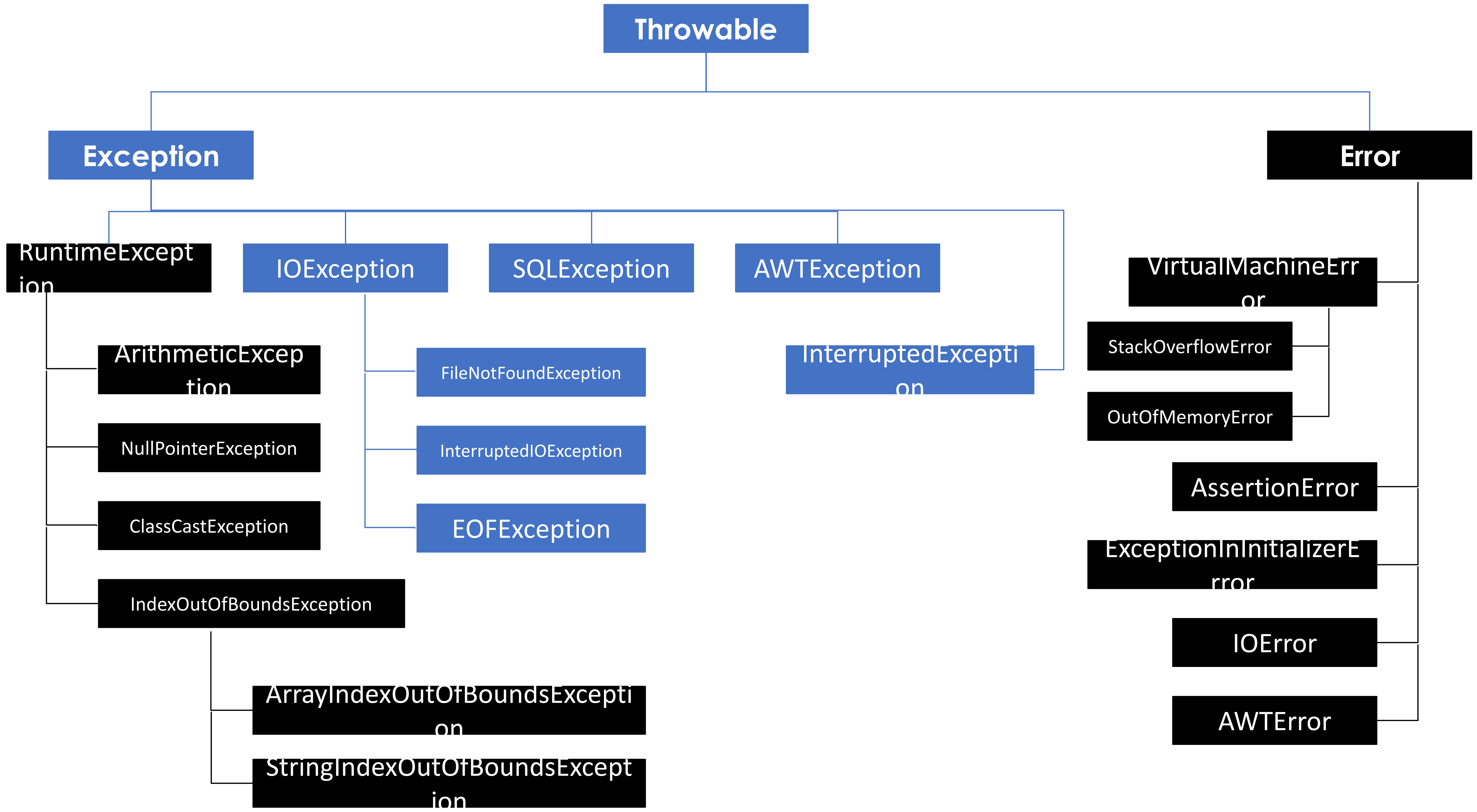
Example : FileNotFoundException

UNCHECKED EXCEPTION

UNCHECKED EXCEPTION :

The compiler-unaware exception is known as the unchecked exception. i.e., the Compiler doesn’t know the statements which are responsible for abnormal situations (Exception). Hence, the compiler will not force the programmer either to handle or declare the exception.

Example : ArithmeticException


Throwable
Exception
Error
RuntimeException
IOException
SQLException
AWTException
InterruptedException
ArithmeticException
NullPointerException
ClassCastException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
FileNotFoundException
InterruptedIOException
EOFException
VirtualMachineError
StackOverflowError
OutOfMemoryError
AssertionError
ExceptionInInitializerError
IOError
AWTError

NOTE :

- **In Throwable hierarchy Error class and its subclasses, RuntimeException class and its subclasses are all considered as Unchecked Exceptions.**
- **All the subclasses of the Exception class except RuntimeException are considered as Checked Exceptions.**
- **Throwable and Exception classes are partially checked and partially unchecked.**

THROWABLE

Throwable :

Throwable class is defined in java.lang package.

NOTE :

In the Throwable class, all the built-in classes of the Throwable type are overridden toString() method such that it returns the fully qualified name of the class.

Important methods of Throwable class :

- String getMessage();
- void printStackTrace();

EXCEPTION HANDLING

EXCEPTION HANDLING :

Exception handling is a mechanism used in java that is used to continue the normal flow of execution when the exception occurred during runtime.

HOW TO HANDLE THE EXCEPTION?

In java, we can handle the exception with the help of a try-catch block.

Syntax to use try catch block :

```
try
{
        // Statements ;
}
catch(declare one variable of Throwable type)
{
        // Statements ;
}
```

try-catch BLOCK

try{ }:

- The statements which are responsible for exceptions should be written inside the try block.
- When an exception occurs,

  I. **Execution of try block is stopped.**
  II. **A Throwable type object is created.**
  III. **The reference of throwable type object created is passed to the catch block.**

EXAMPLE :

```
        try
        {
                stmt1;
                stmt2;   //Exception occurs
                stmt2;
                stmt3;
        }
        catch(variable)
        {

        }
```

stmt3 and stmt4 will not execute

0x1

Throwable type object is created

Reference of Throwable type object is thrown to the catch block

catch() { } :

The catch block is used to catch the throwable type reference thrown by the try block.

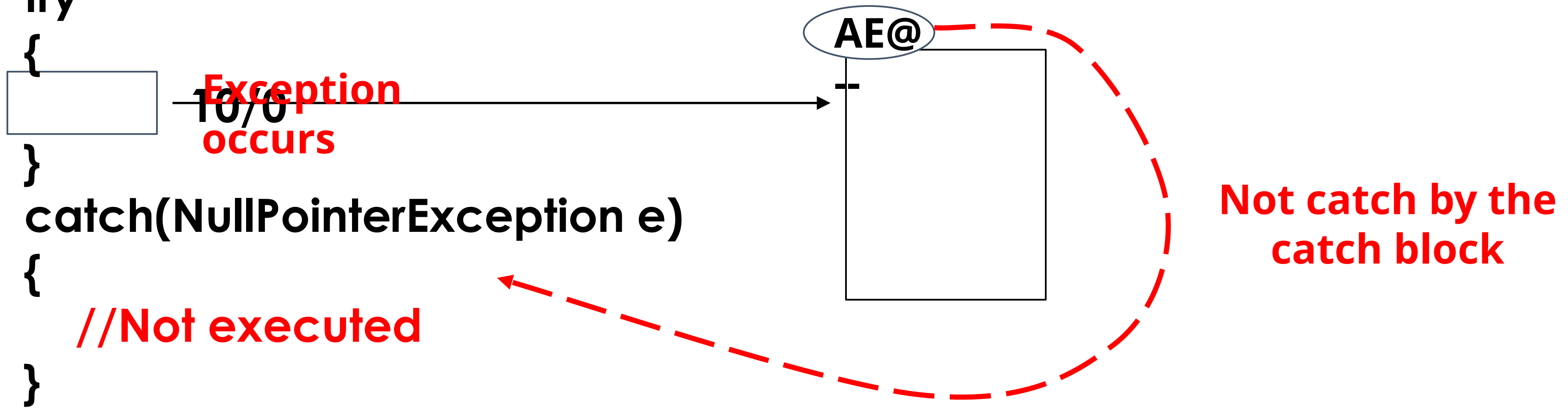
1. If it catches, We say the exception is handled. Statements inside the catch block are get executed and the normal flow of the program will continue.
2. If it doesn't catch, we say the exception is not handled. Statements written inside the catch block are not executed and the program is terminated.

When do we say exception is handled ?

We say Exception is handled only if Exception is caught by catch block.

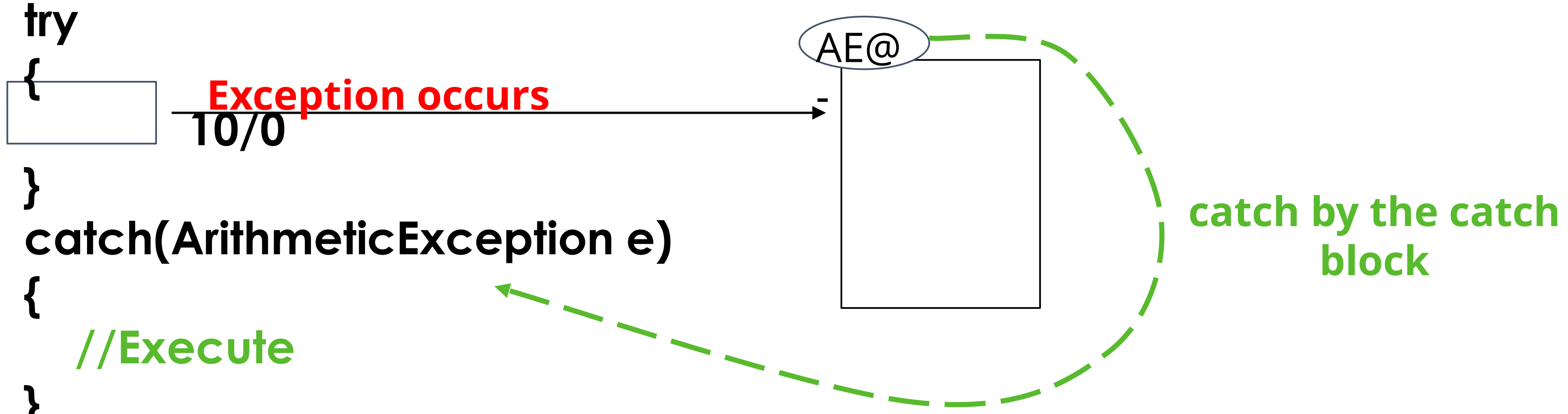
Case 1 : Exception occurs not caught :

```
try
{
   10/0
}
catch(NullPointerException e)
{
   //Not executed
}
//Not executed
```



Case 2 : Exception occurs and caught :

```
try
{
        10/0
}
catch(ArithmeticException e)
{
    //Execute
}
//Execute
```



EXAMPLE :

EXCEPTION OCCURS BUT NOT HANDLED

```
System.out.println("Main Begins");
try()
{
   int c = 5/0;
}
catch(NullPointerException e)
{
System.out.println("Divisible by zero is not
possible");
}
System.out.println("Main End");
```

CONSOLE :

```
   Main Begins
   Exception in thread "main"
java.lang.ArithmeticException: / by zero
```

EXCEPTION OCCURS AND HANDLED

```
System.out.println("Main Begins");
try()
{
   int c = 5/0;
}
catch(ArithmeticException e)
{
System.out.println("Divisible by zero is not
possible");
}
System.out.println("Main End");
```

CONSOLE :

```
   Main Begins
   Divisible by zero is not possible
   Main End
```

TRY WITH MULTIPLE CATCH BLOCK

try with multiple catch :

try block can be associated with more than one catch blocks

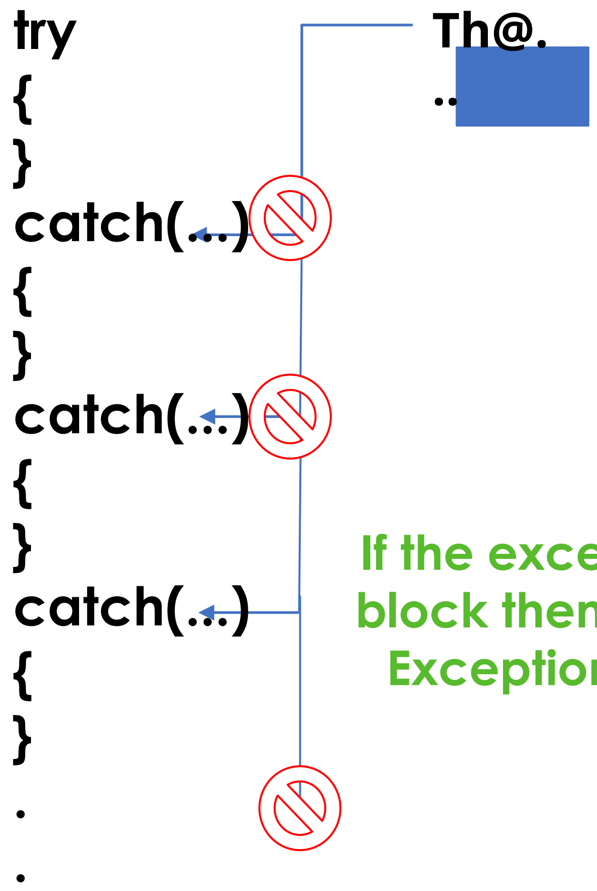
Syntax :

```
try
{
}
catch(...)
{
}
catch(...)
{
}
.
.
.
```

NOTE :

The Exception type object thrown from top to bottom order.

WORKFLOW :



```
try
{
}
catch(...)
{
}
catch(...)
{
}
catch(...)
{
}
.
.
```

If the exception is caught by the catch block then try block does not throw the Exception object to the below catch blocks

RULE :

The order of catch block should be maintained such that, child type should be on the top and parent type at the bottom.

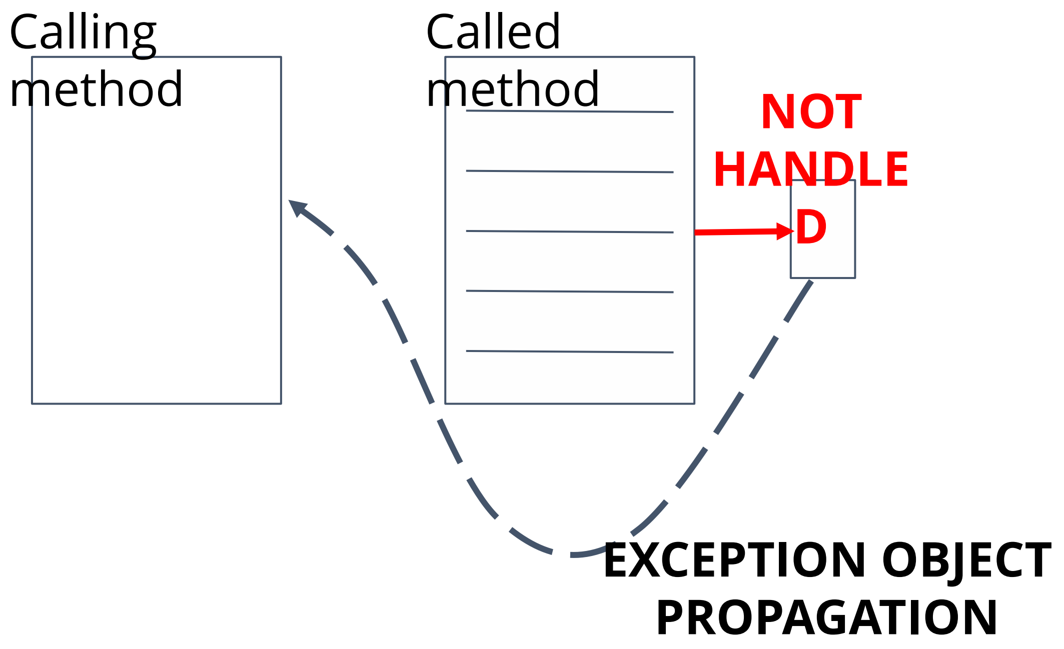
EXAMPLE :

CASE 1	CASE 2
try	try
{	{
10/0;	10/0;
}	}
catch(Exception e)	catch(ArithmeticException e)
{	{
}	}
catch(ArithmeticException e)	catch(Exception e)
{	{
}	}
CTE : Parent type is declared on the top and child type is on bottom	CTS : Parent type is declared on the bottom and child type is on top

EXCEPTION OBJECT PROPAGATION

EXCEPTION OBJECT PROPAGATION :

The movement of exception from called method to calling method when it is not handled is known as Exception object propagation.

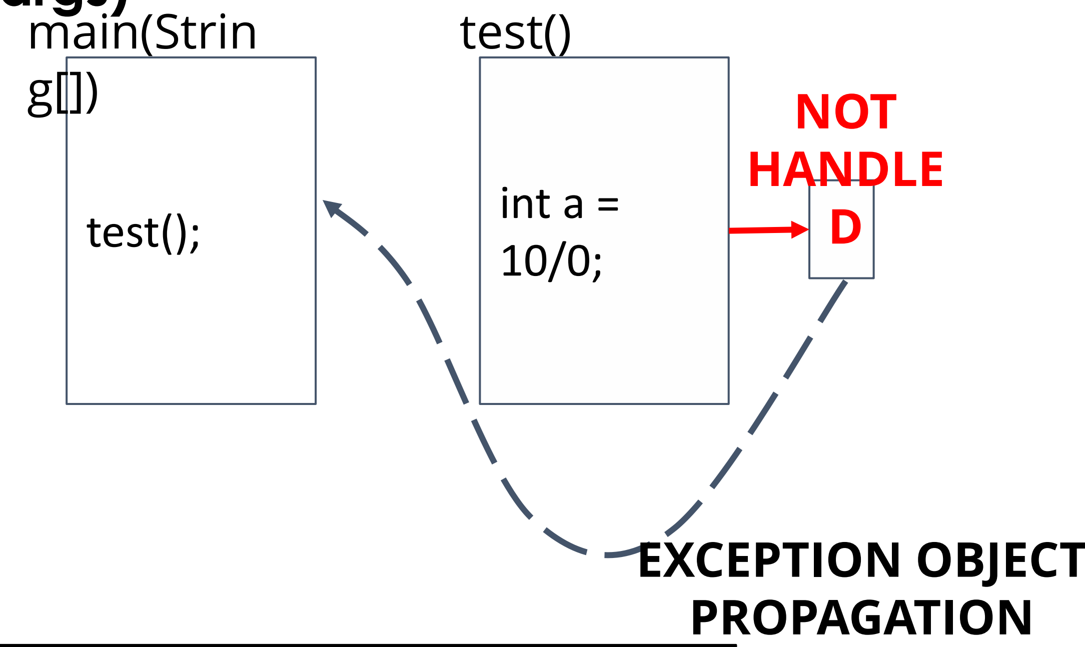


CASE : EXCEPTION OCCURRED AND HANDLED BY THE CALLING METHOD :

```
class Case1
{
        public static void main(String[] args)
        {
                try
                {
                        test();
    }
    catch(ArithmeticException e)
                {
                        System.out.println("Exception is handled by the
calling method");
                }
        }
        static void test()
        {
                Int a = 10/0;
        }
}
```

CASE : EXCEPTION OCCURRED AND NOT HANDLED BY THE CALLING METHOD:

```
class Case1
{
        public static void main(String[] args)
        {
                test();
        }
        static void test()
        {
                Int a = 10/0;
        }
}
```



```
Exception in thread “main” java.lang.ArithmeticException: / byzero
        at Case1.test(Case1.java: 8)
        at Case1.main(Case1.java: 4)
```

CHECKED EXCEPTION

CHECKED EXCEPTION :

- The compiler-aware exception is known as checked exception.
- If any statements are responsible for checked exception it is mandatory to either declare or handle the exception otherwise, we will get unreported compile time error.
- Compiler will force the programmer to either declare or handle the checked exception during compile time.

Example: FileNotFoundException

FileNotFoundException :

- FileNotFoundException is defined in the java.io package.
- We get FileNotFoundException when we try to create a file but the given path is wrong or no permission or there is not sufficient memory in the hard disk.
- **new FileOutputStream("path/name")** ---> this statement is responsible for
- FileNotFoundException.
- FileNotFoundException is a **checked exception.**

EXAMPLE 1:

```
        import java.io.FileOutputStream;
    class FileNotFoundDemo1
        {
                public static void main(String[] args)
                {
                        // to create a file demo.txt in e://f1
                        FileOutputStream fout = new
FileOutputStream("e://f1/demo.txt");
                        System.out.println("File is created");
                }
        }
```

We will get CTE, because new FileOutputStream("e://f1/demo.txt") is responsible for FileNotFoundException (Checked Exception), it is neither declared nor handled.

EXAMPLE 2: Resolving Compile Time Error by handling checked exception

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
class FileNotFoundDemo2
    {
            public static void main(String[] args)
            {
                    // to create a file demo.txt in e://f1
                    try
                    {
                            FileOutputStream fout = new
FileOutputStream(“e://f1/demo.txt”);
                            System.out.println(“File is created”);
                    }
                    catch(FileNotFoundException e)
                    {
                            System.out.prinltn(e);
                    }
                    System.out.printn(“Main end”);
            }
    }
```

THROW KEYWORD

throw :

- It is a keyword.
- It is used to throw an exception manually.
- By using throw we can throw checked exception, unchecked exception. It is mainly used to throw the custom exception.

SYNTAX :

throw exception ;

throw new CustomException(“String”);

EXAMPLE :

```
        class throwDemo
        {
                public static void main(String[] args)
                {
            int a = 15;
            int b = 10;
            if(a>b)
                throw new ArithmeticException("manually thrown");
            else
                System.out.println("No exception");
                }
        }
```

OUTPUT :

```
Exception in thread main java.lang.ArithmeticException: manually thrown
```

NOTE :

If we don't handle the manually thrown exception then we will get unreported exception.

CUSTOM EXCEPTION :

```
class NotValidException extends Exception
{
        NotValidException(String s)
        {
                super(s);
        }
}
```

Raising and handing the NotValidException

```
class Driver
{
        public static void main(String[] args)
        {
                try
                {
                        throw new NotValidException();
                }
                catch(NotValidException e)
                {
                        System.out.println(“Custom exception is handled”);
                }
        }
}
```

THROWS KEYWORD

throws :

- It is a keyword.
- It is used to declare an exception to be thrown to the caller.

NOTE :

throws keyword should be used in the method declaration statement.

SYNTAX :

```
        [modifier] return_type methodName([formal arg]) throws excep1,
excep2,...
        {
                //stmts;
        }
```

NOTE :

If a method declares an exception using throws keyword, then caller of the method must handle or throw the exception. If not we will get compile time error.

EX

DECLARATION	PURPOSE
public static void sleep(long) throws InterruptedException	The purpose of this method is to pause the execution of a program to the specified time.

EXAMPLE 2: InterruptedException is not either handled or declared

```
class throwsDemo1
{
        public static void main(String[] args)
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println("Hello");
                        Thread.sleep(5000);
                }
        }
}
```

We get CTE, because Thread.sleep(long) method declares InterruptedException (Checked Exception). main(String[] args) is a caller of Thread.sleep(long) method but it is not declaring or handling the InterruptedException.

EXAMPLE 2: InterruptedException handled by main method

```
class throwsDemo2
{
	public static void main(String[] args)
	{
		for(int i=1;i<=5;i++)
		{
			System.out.println("Hello");
			//
			Try

			{
				Thread.sleep(5000);
			}
			catch(InterruptedException e)
			{
				System.out.println(e);
			}
		}
	}
}
```

EXAMPLE 2: InterruptedException instead of handling again thrown by main method

```
class throwsDemo2
{
	public static void main(String[] args) throws InterruptedException
	{
		for(int i=1;i<=5;i++)
		{
			System.out.println("Hello");
			Thread.sleep(5000);
		}
	}
}
```

NOTE :

throws keyword does not handle the exception

FINALLY BLOCK

finally { }

- It is a block which is used along with try-catch block.
- It will execute even the exception is not handled

Syntax :

```
try
{
}
catch(...)
{
}
finally
{
}
```

NOTE :

We can also use finally block with try block alone.

FINAL VS FINALLY

final	finally{ }
final is a modifier	finally is a block
It is applicable to class, method, variable. final class can't be inherited, final variable value can't be changed , final method can't be inherited	Instruction written inside the 'finally block' will be executed even if the exception isn't handled.

WRAPPER CLASS

WRAPPER CLASS :

- The wrapper class in java provides mechanism to wrap the primitive into an object.
- For every primitive data type corresponding class is declared known as a wrapper class.
- There are eight wrapper classes declared in java.lang package which provides several methods to convert primitive into an object.

NOTE :

Since J2SE 5.0 Conversion of primitive to Object and vice versa is implicitly done by the compiler.

WRAPPER CLASSES

PRIMITIVE DATA TYPES	WRAPPER CLASSES
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

NOT

Among these wrapper classes Byte, Short, Integer, Long, Float, Double are subclasses of Number class.

AUTOBOXING

AUTOBOXING :

The process of implicitly converting a primitive data type into corresponding wrapper class(Object) is known as autoboxing.

EXAMPLE :

int-Integer, byte-Byte, boolean-Boolean, etc.,

EXAMPLE :Autoboxing of int to Integer type

```
class Case1
{
	public static void main(String[] args)
	{
			int i = 10;
			Integer Obj = i ; //Autoboxing
	}
}
```

AUTO UNBOXING

AUTO UNBOXING :

The process of implicitly converting a object into primitive data type is known as auto unboxing.

EXAMPLE

Integer-int, Byte-byte, Boolean- boolean, etc.,

EXAMPLE: AutoUnboxing of Integer to int type

```
class Case1
{
	public static void main(String[] args)
	{
			Integer obj = 10;
			int i = obj ; //Auto Unboxing
	}
}
```

valueOf() METHOD

valueOf() Method :

We can wrap a primitive value to corresponding Wrapper class object by using a valueOf() method.

Declaration :

```
public static wrapper Value valueOf(String);
public static wrapper Value valueOf(primitive data);
```

EXAMPLE :

```
class Demo
{
        Public static void main(String[] args)
        {
                        //primitive data
                        byte b = 10;
                        short s = 20;
                int i = 30;
                long l = 40;
                float f = 10.0f;
                double d = 20.05;
                char c = ‘a’;
                        //converting primitive to object
                        Byte obj1 = Byte.valueOf(b);
                        Short obj2 = Short.valueOf((s);
                        Integer obj3 = Integer.valueOf(i);
                        Long obj4 = Long.valueOf(l);
                        Float obj5 = Float.valueOf(f);
                        Double obj6 = Double.valueOf(d);
                        Character obj7 = Character.valueOf(c); //CTE
                        String str = String.valueOf(i);
        }
}
```

[primitiveDataType]Value() METHOD

primitiveDataType**Value() Method :**

This method is used to find the primitive value for given Wrapper Object.

Declaration :

```
public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
```

NOTE :

- **It is declared inside all the six sub classes of a Number class(Byte, Short, Int, Long, Float, Doble).**
- **Additional to this Character class has charValue() method and Boolean class has booleanValue() method.**

EXAMPLE 1:

```
class Demo1
{
        public static void main(String[] args)
{
        // Object
        Long i = 20l;
        // Converting Object to primitive value
        byte b = i.byteValue();
        short s = i.shortValue();
        int in = i.intValuelong();
    long l = i.longValue();
        float f = i.floatValue();
        double d = i.doubleValue();
}
}
```

EXAMPLE 2:

```
class Demo2
{
        public static void main(String[] args)
{
        // Object
        Character obj = 'c';
        // Converting Object to primitive value
        char ch = obj.charValue();


}
}
```

EXAMPLE 3:

```
class Demo3
{
        public static void main(String[] args)
{
        // Object
        Boolean obj = false;
        // Converting Object to primitive value
        boolean b = obj.booleanValue();


}
}
```

parse[PrimitiveDataType]() METHOD

parse[PrimitiveDataType]**() Method :**

This method is used to convert a given string into a primitive value except the character.

Declaration :

Byte	: public static byte parseByte(String) throws NumberFormatException;
Short	: public static short parseShort(String) throws NumberFormatException;
Integer	: public static int parseInt(String) throws NumberFormatException;
Long	: public static long parseLong(String) throws NumberFormatException;
Float	: public static float parseFloat(String) throws NumberFormatException;
Double	: public static double parseDouble(String) throws NumberFormatException;
Boolean	: public static boolean parseBoolean(String)

NOTE :

The runtime string should be of number format, otherwise we will get NumberFormatException during runtime.

EXAMPLE 1:

```
class Demo
{
	public static void main(String[] args)
	{
			String str = "10";
			byte b = Byte.parseByte(str);
			short s = Short.parseShort(str);
			int i = Integer.parseInt(str);
			float f = Float.parseFloat(str);
			double d = Double.parseDouble(str);
	}
}
```

EXAMPLE 2: Raising NumberFormatException

```
class Demo
{
    public static void main(String[] args)
    {
            String str = "ten";
            byte b = Byte.parseByte(str);
            short s = Short.parseShort(str);
            int i = Integer.parseInt(str);
            float f = Float.parseFloat(str);
            double d = Double.parseDouble(str);
    }
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "ten"
        at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Byte.parseByte(Byte.java:149)
        at java.lang.Byte.parseByte(Byte.java:175)
```

EXAMPLE : public static boolean parseBoolean(String);

```
class Demo
{
	public static void main(String[] args)
	{
			String s = "True";
			boolean b = Boolean.parseBoolean(s);
			System.out.println(b); //true
	}
}
```

NOTE :

If any data passed other than a boolean in a parseBoolean() as a String, then the method will return a false.

COLLECTION FRAMEWORK

Why do we need collection Framework in java ?

To store multiple objects or group of objects together we can generally use arrays.But arrays has some limitations,

LIMITATION OF AN ARRAY

Limitations of an Array :

1. The size of the array is fixed, we cannot reduce or increase dynamically during the execution of the program.
2. Array is a collection of homogeneous elements.
3. Array manipulation such as :
   a. **removing an element from an array.**
   b. **adding the element in between the array etc...**

Requires complex logic to solve.

Therefore, to avoid the limitations of the array we can store the group of objects or elements using different data structures such as :

1. **List**
2. **Set**
3. **Queue**
4. **maps / dictionaries**

Def : Collection Framework is set of classes and interfaces (hierarchies), which provides mechanism to store group of objects (elements) together.

CRUD OPERATION

It also provides mechanism to perform actions such as :

1. create and add an element
2. access the elements
3. remove / delete the elements
4. search elements
5. update elements
6. sort the elements

(CRUD - operations)

Important hierarchies of collection framework

Collection framework has two important hierarchy,

1. **Collection hierarchy**
2. **Map hierarchy**

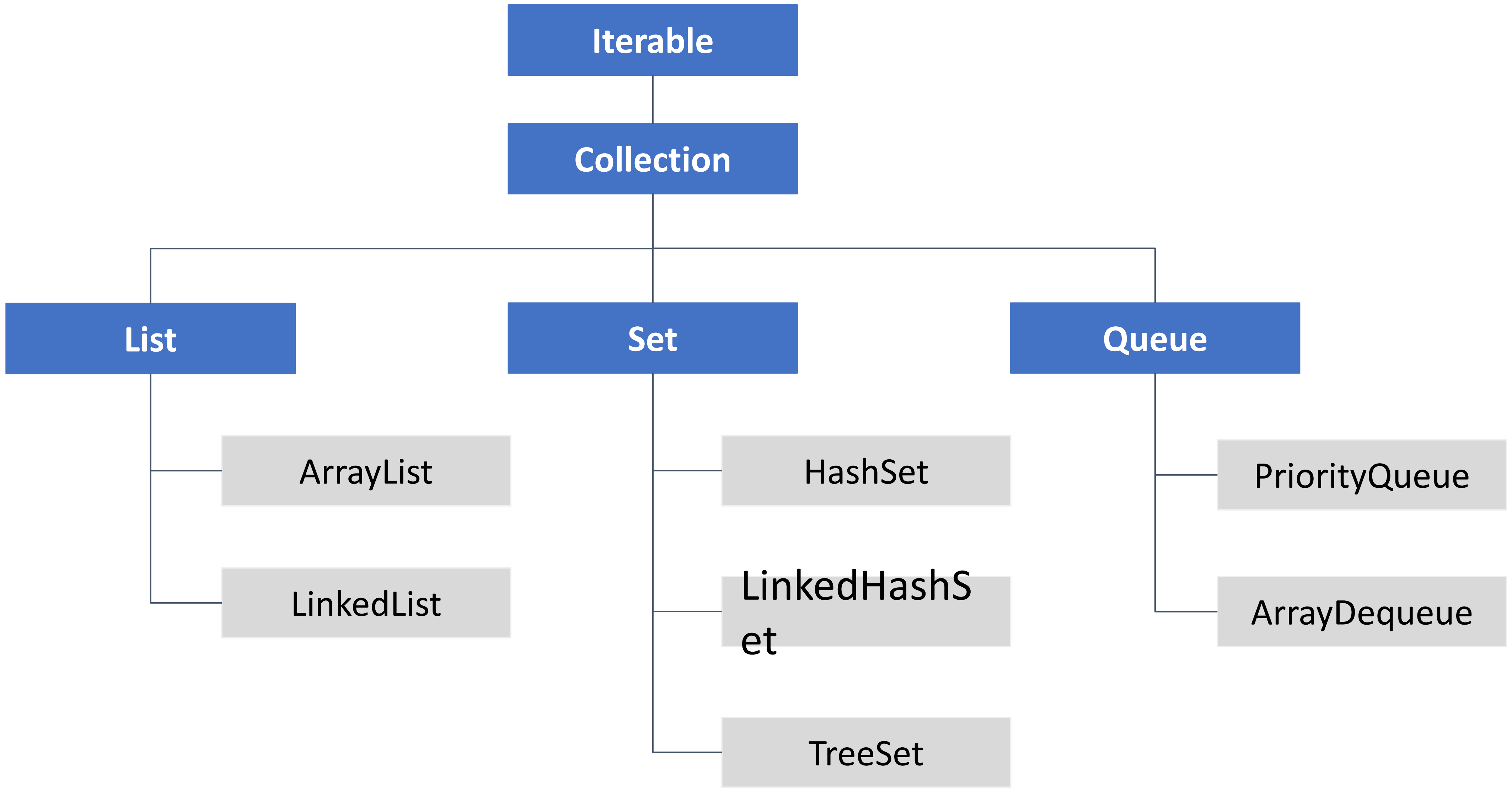
Collection interface

Collection interface :

1. Collection is an interface defined in java.util.
2. Collection interface provides the mechanism to store group of objects(elements) together.
1. All the elements in the collection are stored in the form of Objects. (i.e.only Non-primitive is allowed)
2. which helps the programmer to perform the following task.

a. add an element into the collection
b. search an element in the collection
c. remove an element from the collection
d. access the elements present in the collection

Collection hierarchy



IMPORTANT METHODS OF COLLECTION INTERFACE :

PURPOSE	RETURN TYPE	METHODS
To add an element	boolean	add(Object)
		addAll(Collection)
To remove an element	boolean	remove(Object)
		removeAll(Collection)
		retainAll(Collection)
	void	clear()
To search an element	boolean	contains(Object)
		containsAll(Collection)
To access an element	Iterator	1.iterator() 2.For each loop

miscellaneous		size()
		isEmpty()
		toArray()
		hashCode()
		equals()

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList and LinkedList.

To instantiate the List interface, we must use :

```
1.List <data-type> list1= new ArrayList();
2.List <data-type> list2 = new LinkedList();
```

The classes that implement the List interface are given below.

ArrayList :

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

LinkedList :

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterato itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

1.Queue<String> q1 = **new** PriorityQueue();
2.Queue<String> q2 = **new** ArrayDeque();

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue :

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.
Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
System.out.println("after removing one element:");
Iterator itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set.

Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1.Set<data-type> s1 = **new** HashSet<data-type>();
2.Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3.Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet :

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items. Consider the following example.

```
import java.util.*;
public class TestJavaCollection7  {
public static void main(String args[])   {
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Java Non-generic Vs. Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.

Let's see the old non-generic example of creating a Java collection.

→ **ArrayList list=new ArrayList();**//creating old non-generic arraylist

Let's see the new generic example of creating java collection

→ **ArrayList<String> list=new ArrayList<String>();**//creating new generic arraylist

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of object in it.
If you try to add another type of object, it gives a *compile-time error*.

Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes. However, there are many differences between ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data	LinkedList is **better for manipulating** data

HASH SET

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

LINKED HASH SET

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is non synchronized.
- Java LinkedHashSet class maintains insertion order.

TREE SET

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

COMPARABLE INTERFACE

COMPARABLE INTERFACE :

Comparable interface is a functional interface(Only one abstract method), since it has only one abstract method.

Method declaration :

public abstract int compareTo(Object o)

STEPS TO MAKE AN OBJECT COMPARABLE TYPE :

STEP 1:

The class must implements java.lang.Comparable interface.

STEP 2:

Override compareTo(Object o) method.

compareTo(Object)

compareTo(Object):

- It is defined in Comparable interface.
- This method is used to compare the current object with passed object.
- The return type is int.

FUNCTION :

- If the value of current object is greater then the passed object it will return positive integer.
- If the value of current object is lesser then the passed object it will return negative integer.
- If the value of current object is equal to the passed object it will return zero.

NOTE :

compareTo() method is called only when the object is of comparable type.

EXAMPLE :

```
class Book implements Comparable
{
        int bid;
        String title;
        double price;
        @Override
        public int compareTo(Object o)
        {
                Book b = (Book)o;
                if(this.price==b.price)
                        return 0;
                else if(this.price>b.price)
                        return 1;
                else
                        return -1;
        }
}
```

- A book object is comparable type.
- The array are collection of book objects can be sorted with the help of built in sort method.
- The built in sort method will sort the objects based on the design provided in compareTo(Object) method.It is known as natural ordering.
- In the above example compareTo(Object) method comparison is done on price of the book.Therefore, the sort method will sort the object based on price only.

DISADVANTAGES OF COMPARABLE

Disadvantage of comparable interface:

- We can sort the object only in one defined order (Natural order)
- If the object is not comparable type, then we can’t sort using built in sort methods.

MAPS

Map is a data structure, which helps the programmers to store the data in the form of key value pairs. Where every value is associated with a unique key.

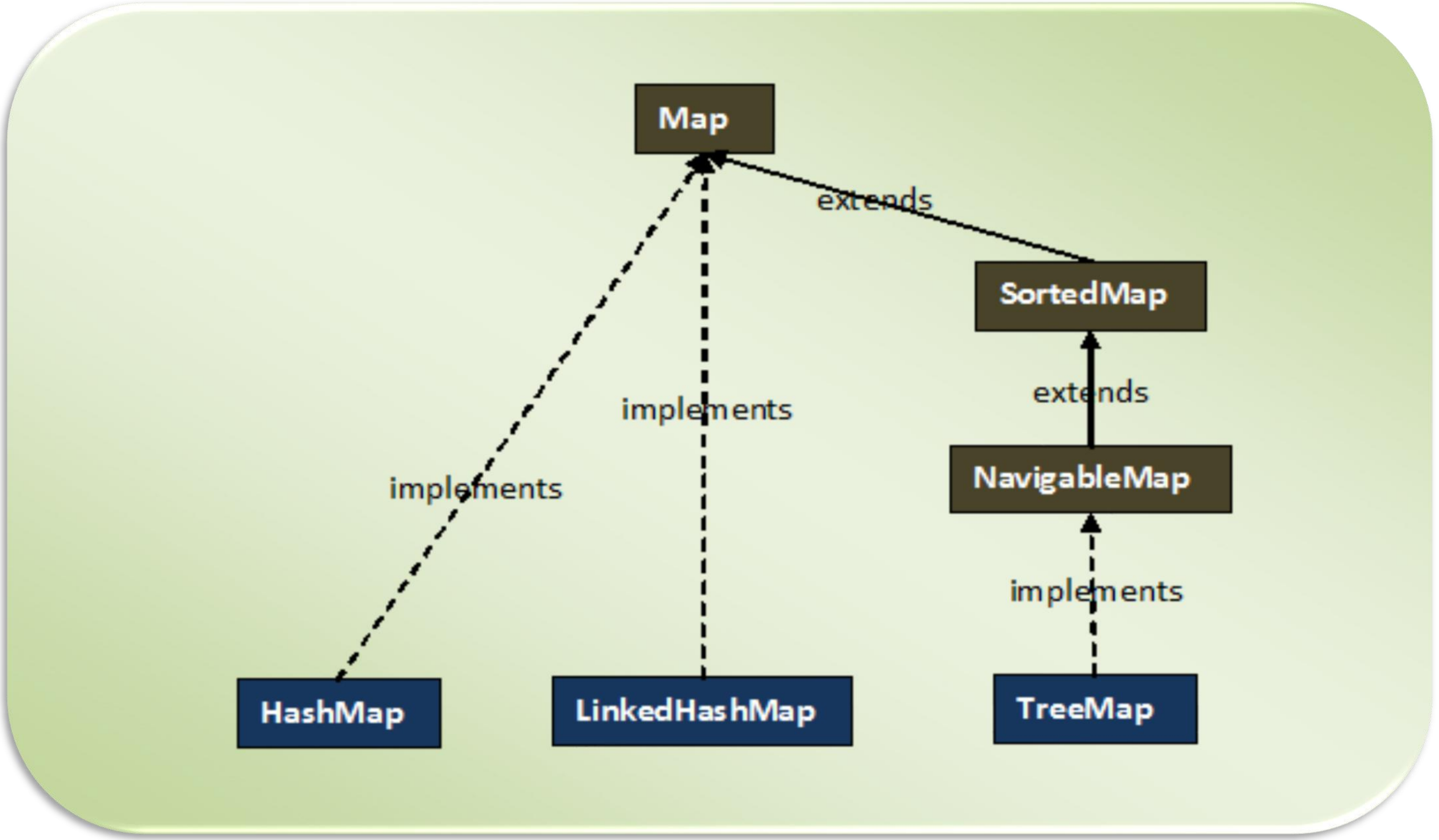
Note :

1. key cannot duplicate.
2. one key can be associated with only one value.

Maps helps us to access the values easily with the help of its' associated key.

Example:

$$\begin{tikzcd}
\text{Key} & \text{value} \\
\text{employee id} \arrow[r] & \text{Employee} \\
\text{pin code} \arrow[r] & \text{City}
\end{tikzcd}$$


Map
extends
SortedMap
extends
NavigableMap
implements
implements
implements
HashMap
LinkedHashMap
TreeMap

1. Map is an interface in java defined in java.util package.
2. We can create generic map by providing the type for both key as well as value, < key_type , value_type >, < K , V >
3. We can obtain 3 different views of a map.
   - a. we can obtain a list of values from a map.
   - b. We can obtain a set of keys from a map.
   - c. We can obtain a set of key-value pairs.

Note : One key value pair is called as an entry or a Mapping.

Methods Of Map interface :

Method Signature	**purpose**
• put(key , value)	1.add an entry to the map (key-value pair) . 2.replace the old value with a new value of an existing entry in the map. putAll(Map) it will copy all the entries from the given map into the current map.
• containsKey(key)	if the key is present returns true else returns false
• containsValue(value)	if the value is present it returns true, else it returns false.
• remove(key)	if the key is present the entry is removed from the map and the value is returned. if the key is not present nothing is removed and null is returned.
• clear()	it removes all the entries in the map.

• get(key)	it is used to access the value associated with a particular key. if the key is present it returns the value. if the key is not present it returns null.
• values()	it returns a collection of values present in the map. return type Collection< v >
• keySet()	it returns a set of keys present in the map return type Set
• entrySet()	it returns a set of all the entries present in the map. return type Set< < k, v> >

- size()
- isEmpty()
- equals()
- hashCode()

Example :

```
public class HashMapDemo {
public static void main(String[] args) {
HashMap<Integer,String> hm=new HashMap();   // generic

// to insert
hm.put(2, "ram");
hm.put(1, "seeta");
hm.put(5, "anil");
hm.put(10, “asha");
hm.put(8, “ravi");

System.out.println(hm);

System.out.println(hm.containsKey(2));

System.out.println(hm.containsValue("seeta"));

hm.remove(1);

System.out.println(hm);

System.out.println(hm.get(5));

System.out.println(hm.values());

System.out.println(hm.keySet());

System.out.println(hm.entrySet());
}
}
```

<u>HashMap :</u>

- It is a concrete implementing class of Map interface.
- data is stored in the form of key=value pair.
- Order of insertion is not maintained.
- key cannot be duplicate, values can be duplicate.
- key can be null.
- value can be null.

<u>TreeMap :</u>

- It is a concrete implementing class of Map interface.
- it also stores data in key=value pair.
- it will sort the entries in the map with respect to keys in ascending order.
- The key in TreeMap must be comparable type. If it is not comparable type we get ClassCastException
- In TreeMap key cannot be null, If it is, null we get NullPointerException.
- In TreeMap key cannot be null, If it is ,null we get NullPointerException.
- A value in TreeMap can be null.

LinkedHashMap

- The **LinkedHashMap Class** is just like HashMap with an additional feature of maintaining an order of elements inserted into it.
- HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

HashTable :

- It is a concrete implementing class of Map interface.
- It is used to store data in key=value format.
- in hashtable the insertion order is not maintained.
- In hashtable both key and value cannot be null. If it is null we get NullPointerException.

ACCESS MODIFIERS

We have two type of modifiers

1. Access modifiers
2. Non access modifiers

Access modifiers :

- Access modifier are responsible to change / modify the accessibility of the member.
- We have four type of access modifiers,

    1. **private**
    2. **default**
    3. **protected**
    4. **Public**

PRIVATE

PRIVATE ACCESS MODIFIER :

- It is a class level modifier , it is applicable for variables , method and constructors.
- If the member of a class is prefixed with private modifier then it is accessible only within the class accessing outside the class is not possible.

EXAMPLE :

```
class A{
private static int i ;
}
class B{
Public static void main(String[] args){
System.out.println(A.i); // CTE
}
```

DEFAULT

DEFAULT ACCESS MODIFIER :

- The accessibility of default modifier is only within the package.It can’t be accessed from outside the package.
- If you don’t declare any access modifier then it is considered as a default access modifier.

EXAMPLE :

```
package myPack ;
public class Demo{
            static int i ;
}
class Driver{
Public static void main(String[] args){
Demo.i = 5 ; // CTE
}
```

PROTECTED ACCESS MODIFIER :

- The access level of a protected modifier is within the package and outside the package through child class.
- If you do not make the child class, it cannot be accessed from outside the package.

PUBLIC ACCESS MODIFIER :

- The access level of a public modifier is anywhere.
- It can be accessed from within the class, outside the class, within the package as well as outside the package.

SCOPE OF AN ACCESS MODIFIER

ACCESS MODIFIER	WITHIN THE CLASS	WITHIN THE PACKAGE	OUTSIDE THE PACKAGE	OUTSIDE THE PACKAGE BY THE CHILD CLASS
private	YES	NO	NO	NO
default	YES	YES	NO	NO
protected	YES	YES	YES	NO
public	YES	YES	YES	YES

The scope of the access modifier based on the accessibility is :

private < default < protected < public

THREAD

THREAD :

- Thread is lightweight subprocess, smallest unit of a processing which is used to execute the program.
- It uses a shared memory area.
- Every thread will have a components which is required for execution.

Example : Stack, Program Counter, etc,.

JAVA PROCESS CONSIST OF

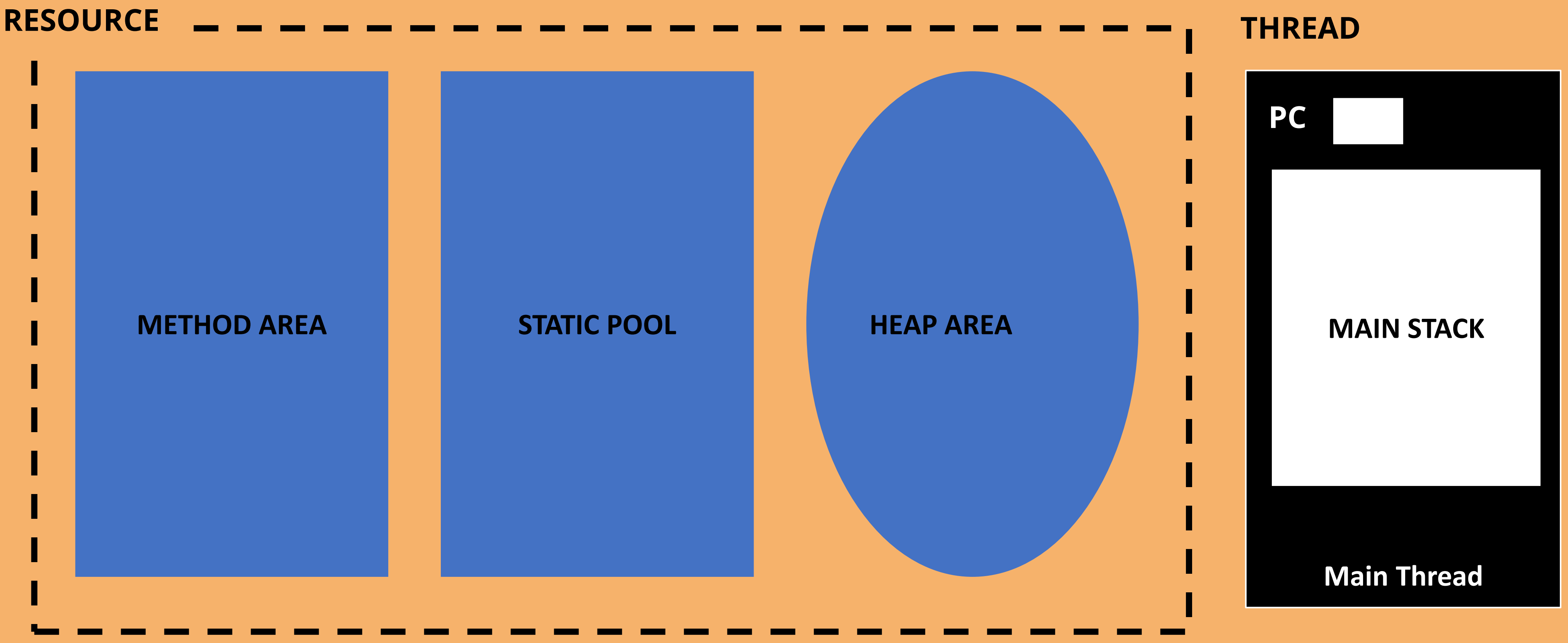
Java process consist of :

- Every process will have default thread for execution, known as Main thread.
- Every process will have a memory to store the resource (Code segment, data segment).

  Code segment - instructions to be stored.

  Data segment - datas are stored.

JAVA RUNTIME EXECUTION PROCESS :



- Every threads will use the common resource.
- main(String[] args) method is always executed by main thread.
- PC-Contains a reference of the next instruction.

MAIN THREAD

MAIN THREAD :

The execution of a main thread always starts from main(String[] args) method and ends at main(String[] args) Method.

Attributes of a thread :

- Name
- Thread ID
- Priority, etc,.

THREAD CLASS

THREAD CLASS :

- In java, we have a class that defines a thread that class is known as Thread class present in java.lang package.
- In thread class name, id, priority, state, etc,. of a thread is defined.
- Thead class has a built in methods to perform actions on thread.
- Thread class is an encapsulated class, all the attributes are made private, we can access them through getter and setter methods.

DECLARATION OF THREAD CLASS:

Thread class declared in java.lang package.

public class Thread extends Object implements Runnable

STATIC METHODS

Return type	Method signature	Function
Thread	currentThread()	Returns the reference of the currently executing thread
boolean	interrupted()	Tests whether the current thread has been interrupted.
void	sleep(long millis)	Cause the currently executing thread to sleep for a specified time.
void	sleep(long millis, int nanos)	Cause the currently executing thread to sleep for a specified time.
void	yield()	A hint to the processor that current thread is willing to yield its current use of a processor.

NON STATIC METHODS

Return type	Method signature	Function
String	getName()	Returns this thread name
long	getId()	Returns this identifier of a thread
int	getPriority()	Returns this thread’s priority
ThreadGroup	getThreadGroup()	Returns thread group to which this thread belongs to
Thread.state	getState()	Returns the state of this thread
void	interrupt()	Interrupts this thread
boolean	isAlive()	Tests if this thread is alive
boolean	isDaemon()	Tests if this thread is daemon
boolean	isInterrupted()	Tests whether this thread is interrupted

NON STATIC METHODS

Return type	Method signature	Function
void	join()	Waits for this thread to die
void	join(long millis)	Waits at most millis millisecond for this thread to die
void	join(long millis, int nanos)	Waits at most millis millisecond plus nanosecond for this thread to die
void	resume()	This method exist for solely for use with suspend()
void	run()	The execution of thread we created is starts from run() method and eds at run().
void	setDaemon(boolean on)	Make this thread as a daemon
void	setPriority(int newPriority)	Changes the priority of this thread
void	start()	Once the start method is called the thread is ready to execute
void	stop()	
void	suspend()	

To access the properties of Main Thread:

- Thread.currentThread()
- getName()
- getPriority()

EXAMPLE:

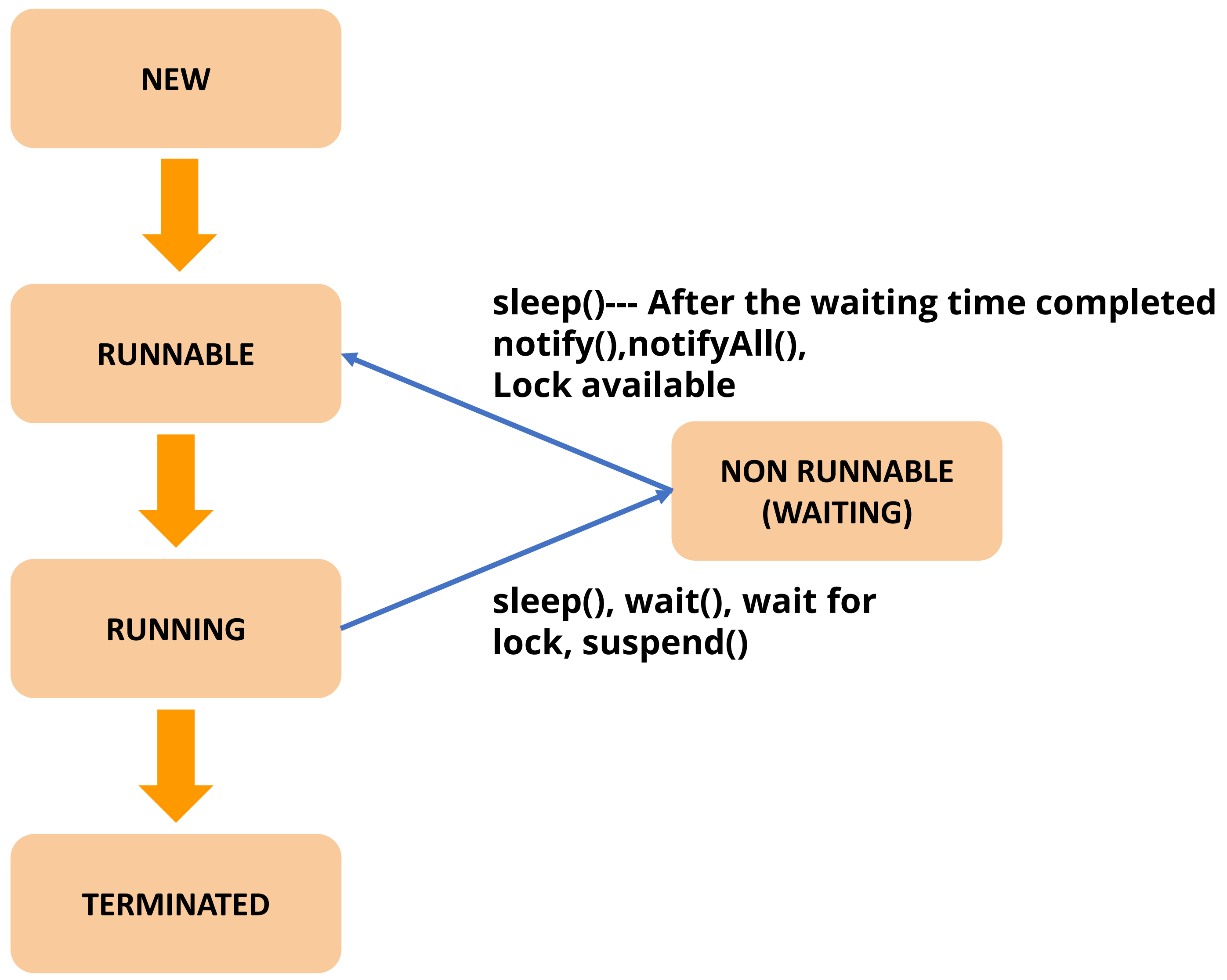
```
class ThreadPropertyDemo
{
        public static void main(String[] args)
        {
                System.out.println(Thread.currentThread().getName());
//main
                System.out.println(Thread.currentThread().getId());

        }
}
```

LIFE CYCLE OF A THREAD

Stages of threads :

- Born **(New)**
- Ready **(Runnable)**
- Executing **(Running)**
- Waiting **(Non runnable)**
- Dead **(Terminated)**


NEW
RUNNABLE
RUNNING
TERMINATED
NON RUNNABLE (WAITING)
sleep()--- After the waiting time completed
notify(),notifyAll(),
Lock available
sleep(), wait(), wait for
lock, suspend()

NEW:

Thread object is created.

RUNNABLE:

By calling start() method thread goes to runnable stage.

RUNNING:

When Thread Scheduler selects the thread for execution we say thread is under execution.

TERMINATED:

- Once execution of the of a thread is successfully completed it goes to dead stage.
- A thread can also be stopped when forcefully send to dead stage.

WAIT STAGE:

- A thread which is under execution if interrupted with the help of methods such as sleep(), wait(, etc,.
- A thread can also goes to wait stage if the required resource is not available.

Java has a beautiful mechanism to create a multiple threads with the help of Thread class.

Why programmer should create a multiple thread ?

- To achieve parallelism
- To reduce a execution time
- Efficient use of resource

BY EXTENDING THREAD CLASS

Creating a thread by extending Thread class :

- Create a class by extending java.lang.Thread class
- Override the run() Method

Executing a thread:

- Inside a main method of any class create an object for the class that extends Thread and overrides runnable.
- Call the start method by using object reference of that instantiated class.

EXAMPLE :

```
class MyThread extends Thread
{
        @Override
        public void run()
        {
                System.out.println(currentThread().getName()+" is executing run()");
                System.out.println("Priority is "+currentThread().getPriority());
        }
        public static void main(String[] args)
        {
                MyThread t = new MyThread();
                t.start();
        }
}
```

JAVA

What is java…?

- Java is a high level object oriented programming language .
- Java was released by sun microsystems in 1995.
- The current version of java is 17 which was released on september 14th 2021.
- Java is very versatile as it is used for developing applications on the web, mobile, desktop, etc. using different platforms. Also, java has many features such as dynamic coding, multiple security features, platform-independent characteristics, etc.

SYLLABUS

- ✓ What is Programming Language.
- ✓ What is JDK,JVM,JRE.
- ✓ TOKEN and types of TOKEN.
- ✓ What is a Variable
- ✓ What is a DataType
- ✓what are Operators
- ✓what is a Method?

- ✓ What is Dynamic reading?
- ✓ What is Control Statements?
- ✓ What are loops?
- ✓ What are Static context?
- ✓ What are non static context?

This will be the end of your PART-1………!!

Concepts in oops

- ✓What is Encapsulation.
- ✓What is Relationships(inheritance).
- ✓ What is Polymorphism.
- ✓ What is Abstraction.
- ✓ What is a Interface.

This is the end of oops (Part-2).......!!!

Part-3

- ✓ What is a Object class.
- ✓ What is a String class.
- ✓ What are Arrays.
- ✓ What is Exception handling.
- ✓ What are Collections.
- ✓ What is Multithreading.

WHAT IS PROGRAMMING LANGUAGE?

- A language or a medium which is used to instruct a computer to perform some particular task is known as Programming language.

- Example: Java, html, C, etc.

TYPES OF PROGRAMMING LANGUAGE:

- 1. High Level languages
- 2. Mid Level languages
- 3. Low Level languages

LOW LEVEL LANGUAGE:

- The language which is easily understandable by the processor is known as Machine level language or low level language
- Binary language is the example of Low level language, i.e.. 0’s and 1’s.

ASSEMBLY LEVEL LANGUAGE

- The Machine level language consists of some predefined words known as mnemonics.

- It is an intermediate level language as it is understandable to some extent by the processor as well by humans.

- Assembler is used to convert assembly level language into machine understandable language.

HIGH LEVEL LANGUAGE

➨A language that is easy, readable, and understandable by human is known as High Level Language.

➨Ex: java, python, C, C++ etc.

- Easy,
- Readable,
- Understandable

JDK(JAVA DEVELOPMENT KIT)

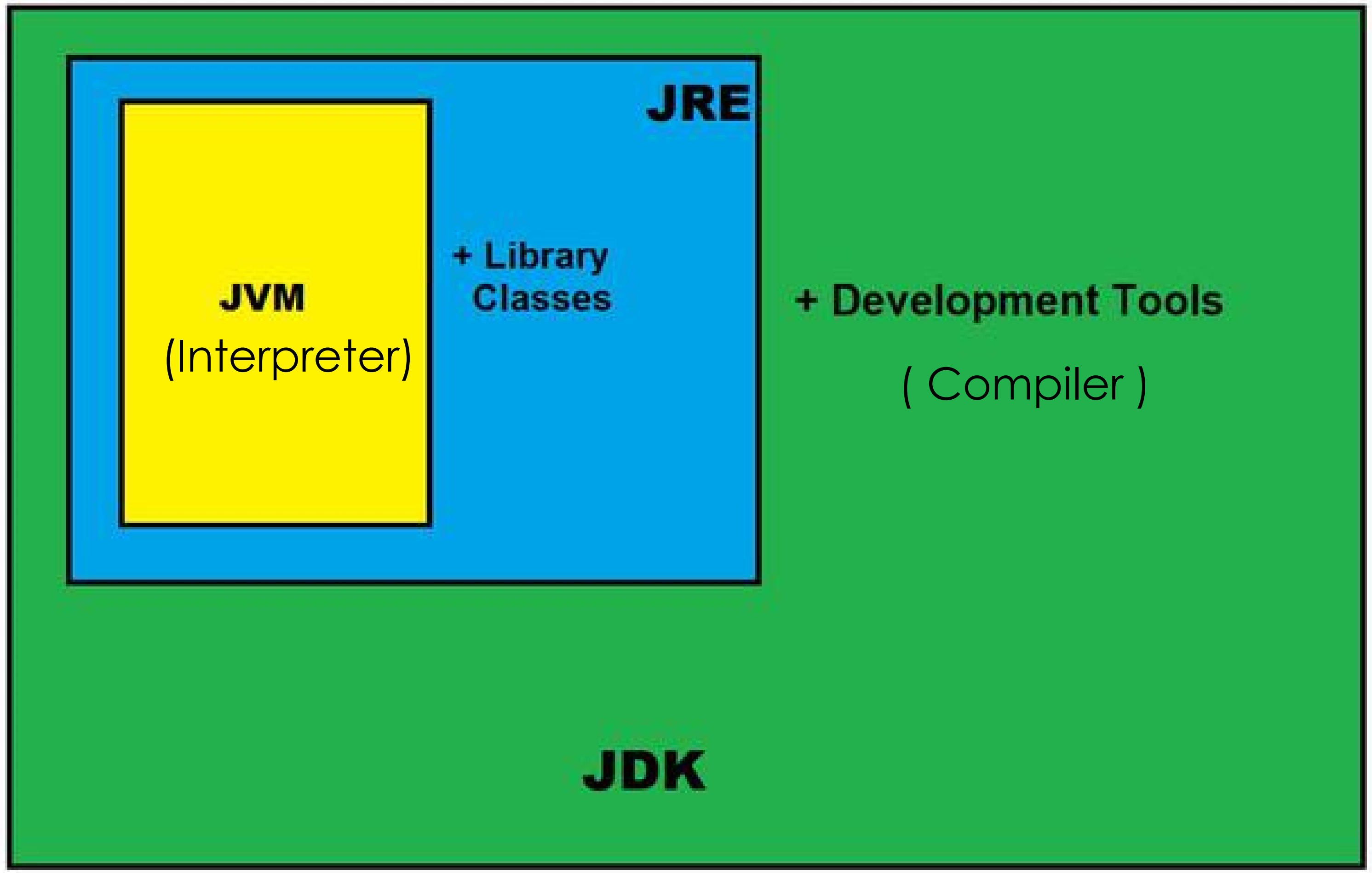
- JDK-is a package which consist of java development tools like java compiler and JRE .

WHAT IS JRE ?

- JRE is an environment which consist of JVM and built in class which is required for the execution of java program.

WHAT IS JVM?

- JVM is responsible for actual execution of a java program.
- It converts the class files into low level language with help of interpreter.


JRE
JVM
(Interpreter)
+ Library Classes
+ Development Tools
(Compiler)
JDK

PLATFORM DEPENDENT

AND

PLATFORM INDEPENDENT

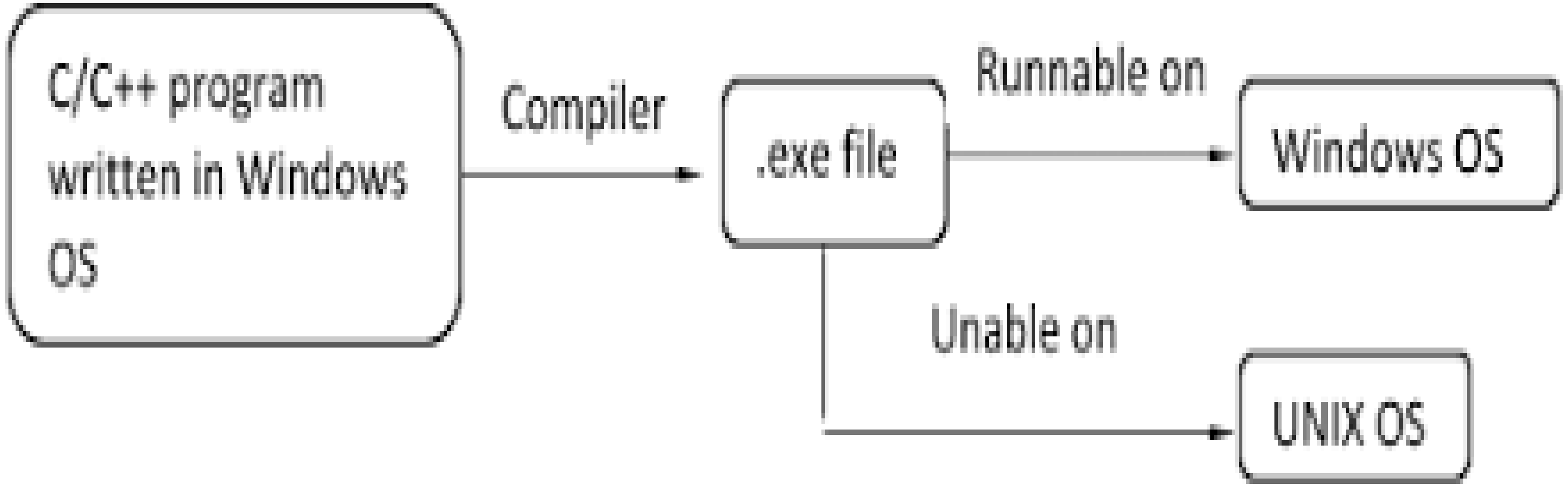
PLATFROM DEPENDENT

- Platform dependent typically refers to applications that run under only one operating system in one series of computers (one operating environment).

- If we develop one application using Windows Operating System, then that application can only be executed on Windows Operating System and cannot be run on other Operating Systems like Mac, Linux etc. This is called platform dependency.

- And the languages used for developing such applications is called Platform Dependent language.

- C and C++ are platform-dependent languages.

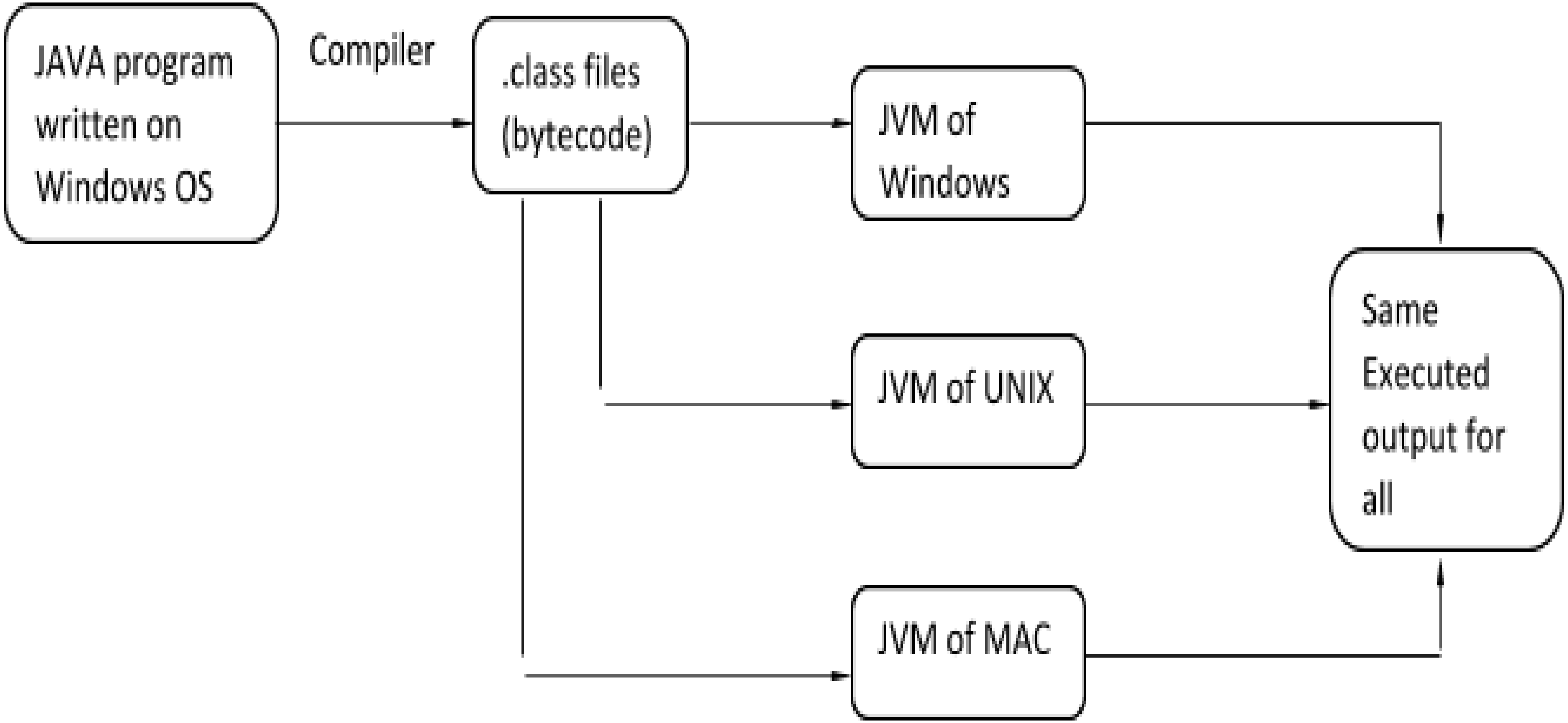

C/C++ program written in Windows OS
Compiler
.exe file
Runnable on
Windows OS
Unable on
UNIX OS

- Platform independent typically refers to applications that works on any platform (operating system) without needing any modification.

- When you write a program in JAVA and compile it, a class file is generated, which consists of byte code. This class file will not be in executable stage.

- The main purpose of generating byte code for a compiled program is to achieve platform independency .

- It means, this byte code generated in one platform(for example, windows OS) can be executed in other platforms such as MAC OS, LINUX OS etc.


JAVA program written on Windows OS
Compiler
.class files (bytecode)
JVM of Windows
JVM of UNIX
JVM of MAC
Same Executed output for all

BASIC STRUCTURE OF A JAVA PROGRAM

- Java instructions are always written inside the class. The syntax is :

```
class  classname
{
        public static void main(String[] args)
        {
                // statements
        }
}
```

Class block

Save file as : classname.java

- Every class in java must have a name. known as class name.
- Every class has a block, known as class block.

- Any class in java can be executed only if main method is created, as shown below :

- Syntax to create a main method :\

```
class Example
{
        public static void main(String[] args)
        {
                // statements
        }
}
```

Main method

Note:

- We can create a class in java without a main method.
- It is compile time successful , so class file will be generated.
- But we cannot execute the class file.

```
class  classname
{
        // statements
}
```

To execute a java program,

Step 1: **Save** the program with the same name as class name (Recommended) with .java extension.

Program1.java

Step 2: Compile the java program using the command –

```
javac filename.java
```

Step 3: Execute the java program using the command –

```
java filename
```

Example program 1:

Write a java program to print hello world

```
class  HelloWorld
{
        public static void main(String[] args)
        {
                System.out.println(“Hello World”);
        }
}
```

OUTPUT :

```
Hello World
```

Example program 2:

Write a java program to print your name and phone number.

```
class  MyDetails
{
        public static void main(String[] args)
        {
                System.out.println(“naMe  : XYZ”);
                System.out.println(“NUMBER  : 123456789”);
        }
}
```

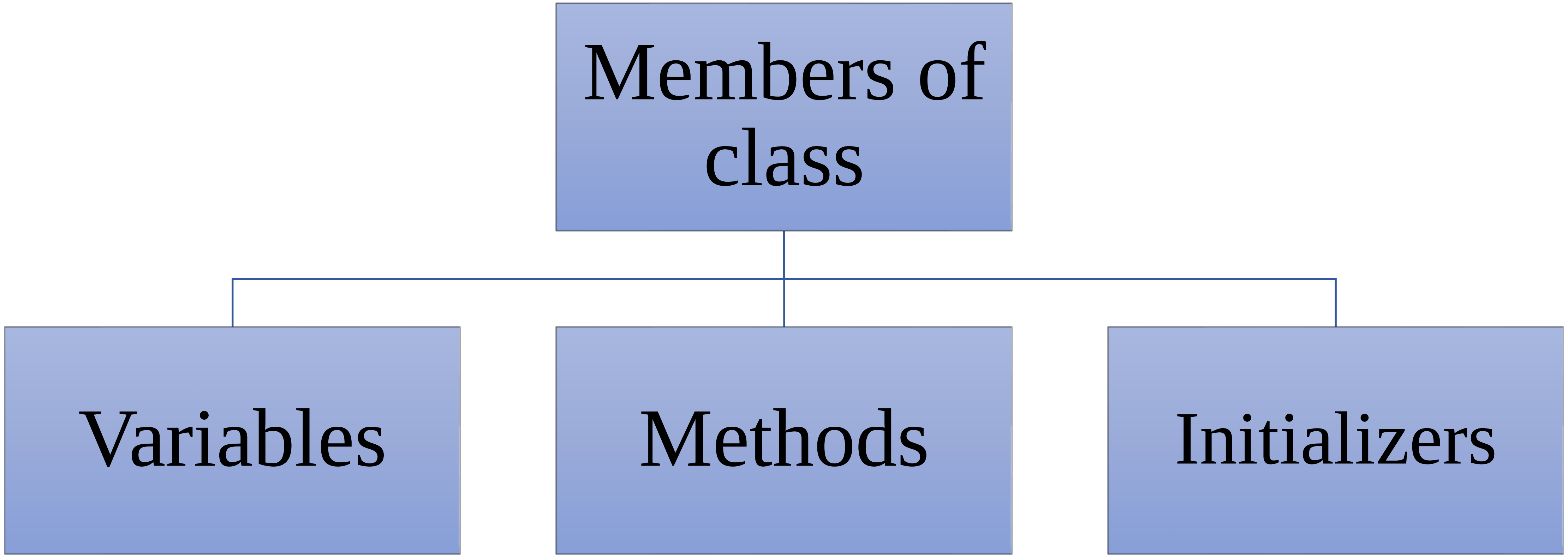
OUTPUT :

```
naMe : XYZ
NUMBER : 123456789
```

Activity 1:

- Write a java program to print hello and good morning.
- Write a java program to print name, percentage and year of pass out.

MEMBERS OF A CLASS



Variables :

- A variable is a container which is used to store data.

Methods :

- A method is a block of instructions which is used to perform a task.

Initializers:

- Initializers are used to execute the start up instructions.

Print statement :

- print statement only prints the data but does not move the cursor to a new line.
- Example : System.out.print(data);
- We cannot use print statement without passing any data.
- System.out.print( ); (Not Allowed)

Println statement :

- println statement prints the data and also moves the cursor to a new line.
- Example : System.out.println(data);
- We can use println statement without passing any data. It will just print new line.
- System.out.println( ); (Allowed)

Program to print Hello and Good Morning using print :

```
class  HelloWorld
{
        public static void main(String[] args)
        {
                System.out.print(“Hello World”);
                System.out.print(“Good Morning”);
        }
}
```

OUTPUT :

```
Hello WorldGood Morning
```

Activity 2:

- Write a java program to print hello and good morning using print statement .
- Write a java program to print your name, percentage and year of pass out using print statement.
- Write a java program having a println statement without passing any data.
- Write a java program having a print statement without passing any data.

TOKENS

In Java, Tokens are the smallest elements of a Java program.

The Java compiler breaks the line of code into text (words) is called **Java tokens**.

For example,

```
class Demo
{
        public static void main(String args[])
        {
                System.out.println(" Hello ");
        }
}
```

In the above code , class, Demo, {, static, void, main, (, String, args, [,],), System, out, ., println, Hello are all the Tokens.

Types of Tokens

Java token includes the following:

- Keywords
- Identifiers
- Literals
- Operators etc.

Keywords

- These are the pre-defined, reserved words which the java compiler can understand.
- Each keyword has a special meaning, i.e.. Each keyword is associated with a specific task.
- A programmer cannot change the meaning of any keyword.
- We have 50+ keywords in java. Few keywords are class, public ,static, void etc.

Note : It is always written in lower case.

Identifier

- It is a name given to the components of java by the programmer.

- Components of java are :
  1. class
  2. variables
  3. methods
  4. interfaces etc.

- Identifiers are defined by the programmer.

Note : A programmer should follow some rules and conventions for defining identifiers.

Rules to define identifiers :

- Identifiers cannot start with number (but may contain numbers).
- It cannot have any special characters except _ and $.
- The whitespace cannot be included in the identifier.
- Identifier name must be different from the keywords (We cannot use keywords as identifiers).

- **Some valid identifiers are** :

1. PhoneNumber
2. PRICE
3. radius
4. a
5. hello123
6. _phonenumber
7. $circumference
8. jagged_array

Conventions

- The coding or industrial standards which are highly recommended to be followed by the programmer is known as conventions.

Note : Compiler will not validate the conventions. Therefore, if conventions aren’t followed, then we will not get compile time error. But it is highly recommended to follow the conventions.

Convention for class name :

- **Single word** : If the class name is a single word, then the first letter must be in upper case and remaining letters in lower case.

  For example, class Addition etc.

- **Multi word** : If the class name is a multi word, then the first letter of each word must be in upper case and remaining letters in lower case.

  For example, class AdditionOfTwoNumbers,

  class SquareRoot, etc.

ACTIVITY 3

1. Write a java program with the class name starting from a number.
2. Write a java program with the class name same as a keyword.
3. Write a java program with the class name starting with $.
4. Write a java program with the class name starting with _.
5. Write a java program with the class name starting with any special character except _ and $.
6. Write a java program with class name as a multi word.

Literals :

- The values or data used in a java program is known as literals.

- The data is generally categorized into two types :
    1. Primitive values
    2. Non Primitive values

- **Primitive values** : Single value data is called as primitive values.

→ Following are the primitive values :

a. **Number literals** (integer number literals (1,4,18 etc), floating number literals (1.0, 19.5, 45.9 etc).

b. **Character literals** (Anything enclosed within single quotes).

c. **Boolean literals** (false, true).

- **Non-Primitive values** : Multi value data (group of data) is known as non primitive value.

- Example of non primitive values are String, Object reference.

- **String literals** : Anything enclosed within double quotes ,i.e.. (“ “) is known as string literals.

→ The length of a string can be anything.

→ Strings are case sensitive.

→Example : “hello”, “TRUE”, “123”, “HelLo”, “Oops@” etc.

Scope of a variable

- Scope of a variable is the part of the program where the variable is accessible.

(or)

- The visibility of a variable is known as Scope of a variable.
- Based on scope of variables, we can categorize variables into three types :

1. Local variables
2. Static variables
3. Non Static variables

Local Variable :

- The variable declared inside a method block or any other block except class block is known as a local variable.

- **Characteristics of a local variable :**

→ We cannot use local variables without initialization. If we try to use local variables without initialization, then we will get compile time error.

→ Local variables will not be initialized with default values.

→ The scope of variables is nested inside the block wherever it is declared. Hence, it cannot be used outside the block.

- **Example :**

```
class Example
{
        public static void main(String[] args)
        {
                int i=10;  // integer type variable i
                float f=10.5;  // float type variable f
                char ch=‘a’;  // character type variable ch
                boolean b=true;  // boolean type variable b
                System.out.println(i);
                System.out.println(f);
                System.out.println(ch);
                System.out.println(b);
        }
}
```

PRIMITIVE DATA TYPES :

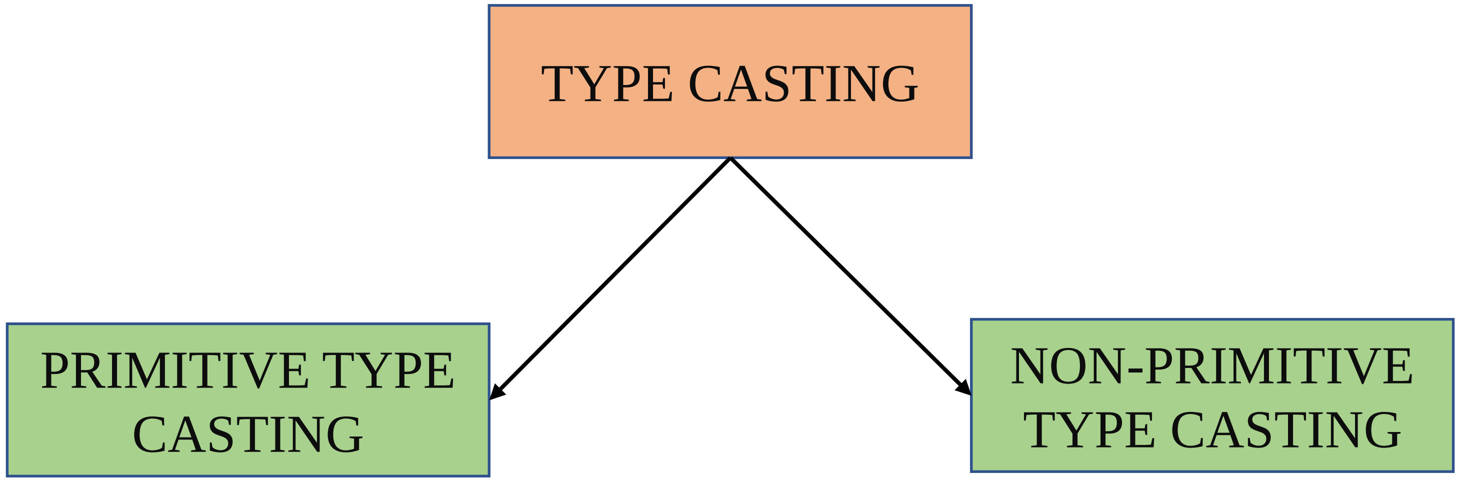
byte	1 byte (stores numbers from -128 to 127)
short	2 bytes
int	4 bytes
long	8 bytes
float	4 bytes
double	8 bytes
boolean	1 bit (true or false) (0 or 1)
char	2 bytes

Byte → short → char → int → long → float → double

TYPE CASTING

What is type casting ?

- The process of converting one type of data is known as type casting.
- There are two types of type casting.



Primitive Type Casting

- The process of converting one primitive type value into another primitive type value is known as primitive type casting.

- There are two types of primitive type casting.

1. **WIDENING**

2. **NARROWING**

WIDENING :

- The process of converting smaller range primitive data type into higher range primitive data type is known as widening.
- In this process, there is no data loss.
- Since there is no data loss, compiler implicitly performs widening. Hence, it is also called as auto-widening.

Byte → short → char → int → long → float → double

Example for widening :

```
class Widening
{
        public static void main(String[] args)
        {
                int a=10;
                float num=a;  // automatically int is converted to float type
                System.out.println(a);
                System.out.println(num);
        }
}
```

Output :

```
10
10.0
```

```
class Widening
{
        public static void main(String[] args)
        {
                int a=10;
                char c=a;  // Compile time error
                System.out.println(a);
                System.out.println(c);
        }
}
```

Note : It will give compile time error since we are trying to convert higher range data type (int) to smaller range data type(char).

NARROWING :

- The process of converting higher range primitive data type into lower range primitive data type is known as widening.
- In this process, there is possibility of data loss.
- Since there is possibility for data loss, compiler will not perform anything implicitly.
- The programmer will have to explicitly(manually) perform Narrowing.
- This can be achieved with the help of type cast operator.

Byte ← short ← char ← int ← long ← float ← double

Example :

```
class Narrowing
{
        public static void main(String[] args)
        {
                int a=10;
                char ch=a;  // Compile time error
                System.out.println(a);
                System.out.println(c);
        }
}
```

Note : It will give compile time error since we are trying to convert higher range data type (int) to smaller range data type(char) without type cast operator.

Type cast operator :

- It is an unary operator.
- It is used to explicitly convert one data type to another data type.

Example for narrowing :

```
class Narrowing
{
	public static void main(String[] args)
	{
		int a=10;
		char ch=( char ) a; //explicitly converting float to int using type cast operator
		System.out.println(a);
		System.out.println(c);
	}
}
```

Output :

```
10.0
10
```

OPERATORS :

Operators are predefined symbols which are used to perform some specific task on the given data.

The data given as input to the operator is known as operand.

Based on the number of operands, there are three types of operators :

→ Unary operator

→ Binary operator

→ Ternary operator

- Based on the task performed, there are 8 types of operators :

  → Arithmetic operators

  → Relational operators

  → Logical operators

  → Assignment operators

  → Conditional operator

  → Bitwise operators

  → Miscellaneous operators

  → Increment/Decrement operators

Ternary operator

- Ternary operator is also known as the conditional operator.
- It is used to handle simple situations in a line.
- It accepts three operands .
- Syntax :

condition ? statement1 : statement2 ;

- The above syntax means that if the value given in condition is true, then statement2 will be executed; otherwise, statement3 will be executed.
- The return type of condition is boolean.
- The return type of conditional operator depends on statement1 and statement2.

Example :

```
int val=5;
String result   =   val==5 ? “you are right” : “you are wrong”;
```

OUTPUT:

```
You are right
```

```
class ConditionalOperator
{
        public static void main(String[] args)
        {
                int a = 6, b = 12;
                int large = a>b ? a :  b;
                System.out.println(" Largest number is : "+large);
        }
}
```

OUTPUT:

```
Largest number is 12
```

INCREMENT / DECREMENT OPERATOR

Increment operator is used to incrementing a value by 1. There are two varieties of increment operator:

- **Post-Increment:** Value is first used for computing the result and then incremented. Example (a++)
- **Pre-Increment:** Value is incremented first and then the result is computed. Example (++a)

Decrement operator is used for decrementing the value by 1. There are two varieties of decrement operators.

- **Post-decrement:** Value is first used for computing the result and then decremented. Example (a--)
- **Pre-decrement:** Value is decremented first and then the result is computed. Example (--a)

PRE INCREMENT / DECREMENT

- We can achieve pre increment or decrement with the help of three steps :

  → Increment / decrement by 1

  → Update the value

  → Substitute the value

Example for pre increment :

```
class  PreInc
{
   public static void main(String[] args)
   {
      int a = 10;
      int b = ++a;
      System.out.println(b);
      System.out.println(++b);

   }
}
```

Output :

```
11
12
```

Example for pre decrement :

```
class  PreInc
{
   public static void main(String[] args)
   {
      int a = 10;
      int b = --a;
      System.out.println(b);
      System.out.println(--b);

   }
}
```

Output :

```
9
8
```

<u>POST INCREMENT / DECREMENT</u>

- We can achieve post increment or decrement with the help of three steps :

  → substitute the value

  → increment / decrement by 1

  → update the value.

Example for post increment :

```
class  PreInc
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = a++;
        System.out.println(b);
        System.out.println(b++);
        System.out.println(b);
    }
}
```

Output :

```
11
11
12
```

Example for post decrement :

```
class  PreInc
{
    public static void main(String[] args)
    {
        int a = 10;
        int b = a--;
        System.out.println(b);
        System.out.println(b--);
        System.out.println(b);
    }
}
```

Output :

```
9
9
8
```

METHODS

- **Method** is a block of instructions which is used to perform some specific task.

- **Syntax to define any method is :**

```
[access modifier] [modifier] returntype methodname ([formal arguments])
{


}
```

It consists of three parts:

1. Method signature
2. Method declaration
3. Method definition

1. Method signature :

- → Method name
- → Formal arguments

2. Method declaration :

- → Access modifier
- → Modifier
- → Return type
- → Method signature

3. Method definition :

- → Method declaration
- → Method body / Method block

MODIFIERS :

- Modifiers are keywords which are responsible to modify the characteristics of the members.

- Example : static, abstract, final, synchronized, volatile, transient etc.

- **static :**
  - →The users can apply static keywords with variables, methods, blocks, and nested classes.
  - →The static keyword belongs to the class .
  - →The static keyword is used for a variable or a method that is the same for every instance of a class.

ACCESS MODIFIER

- Access modifiers are keywords used to change the accessibility of members of class.
- There are four levels of access modifiers :

  → private

  → public

  → protected

  → default

public :

- The public access modifier is specified using the **keyword public**.
- The public access modifier has the widest scope among all other access modifiers.
- Classes, methods, or data members that are declared as public are accessible from everywhere in the program.

- Any method , after execution can return a value back to the caller.
- Therefore, if a method is returning any value, it is mandatory to specify what type of value is returned by the method.
- It must be specified in the method declaration statement.
- This is done with the help of return type.

<u>RETURN TYPE :</u>

- Return type is a data type which specifies what type of data is returned by the method after execution.
- A method can have the following data types :

  → Primitive data types

  → Non-Primitive data types

  → void

void : It is a data type which is used as a return type when the method is returning nothing.

Note : A method can have any number of methods.

Note : A method cannot be created inside another method.

Example :

```
class Example
{
        public static void add()
        {
                int a=10;
                int b=20;
                int sum=a+b;
                System.out.println("sum is : "+sum);
        }
}
```

RETURN STATEMENT :

- A method can return a data back to the caller with the help of return statement.
- **return :** It is a keyword.
- It is a control transfer statement.
- When the return statement is executed, the execution of the method is terminated ant control is transferred back to the calling method.

Steps to use return statement :

→Provide a return type for a method.

→ Use the return statement in the value to be returned.

Note : The type specified as return type must be same as the type of value passed in a return statement.

- A method will get executed only when it is called. We can call a method with the help of method call statement.

<u>**METHOD CALL STATEMENT :**</u>

- The statement which is used to call a method is known as method call statement.
- Syntax to create a method call statement is :

```
                methodName ( [ Actual arguments ]);
```

FLOW OF METHOD CALL STATEMENT :

1. Execution of calling method is paused.
2. Control is transferred to the called method.
3. Execution of called method begins.
4. Once the execution of called method is completed, the control is transferred back to the calling method.
5. Execution of calling method resumes.

Example :

```
class Example
{
        public static void m1()
        {
                System.out.println(“From m1 method“ );
        }
        public static void main(String[] args)
        {
                System.out.println(“start“ );
                m1();
                System.out.println(“end“ );
        }
}
```

Output :

```
start
From m1 method
end
```

TYPES OF METHODS :

- There are two types of method.

  → Parameterized method

  → No argument method

1. **Parameterized method :** The method which consists of formal arguments is known as parameterized method.

   The parameterized methods are used to accept the data.

2. **No argument method :** The method which doesn’t consists of any arguments is known as no argument method.

Formal Arguments : Variables which are declared in method declaration statement are known as formal arguments.

Actual Arguments : The values passed in the method call statement are known as actual arguments.

RULES TO CALL THE PARAMETERIZED METHOD

- The number of actual arguments should be same as the number of formal arguments.

- The type of corresponding actual argument should be same as the type of formal argument.

- If not, the compiler tries to implicitly perform conversion. If it is not possible, then we will get compile time error.

DECISION MAKING STATEMENTS (Control statements)

- Decision making statements help the programmer to skip a block of instruction from execution if the condition is true.

- Types of decision making statements :

  → if statement

  → if else statement

  → if else-if statement

  → switch statement

If statement :

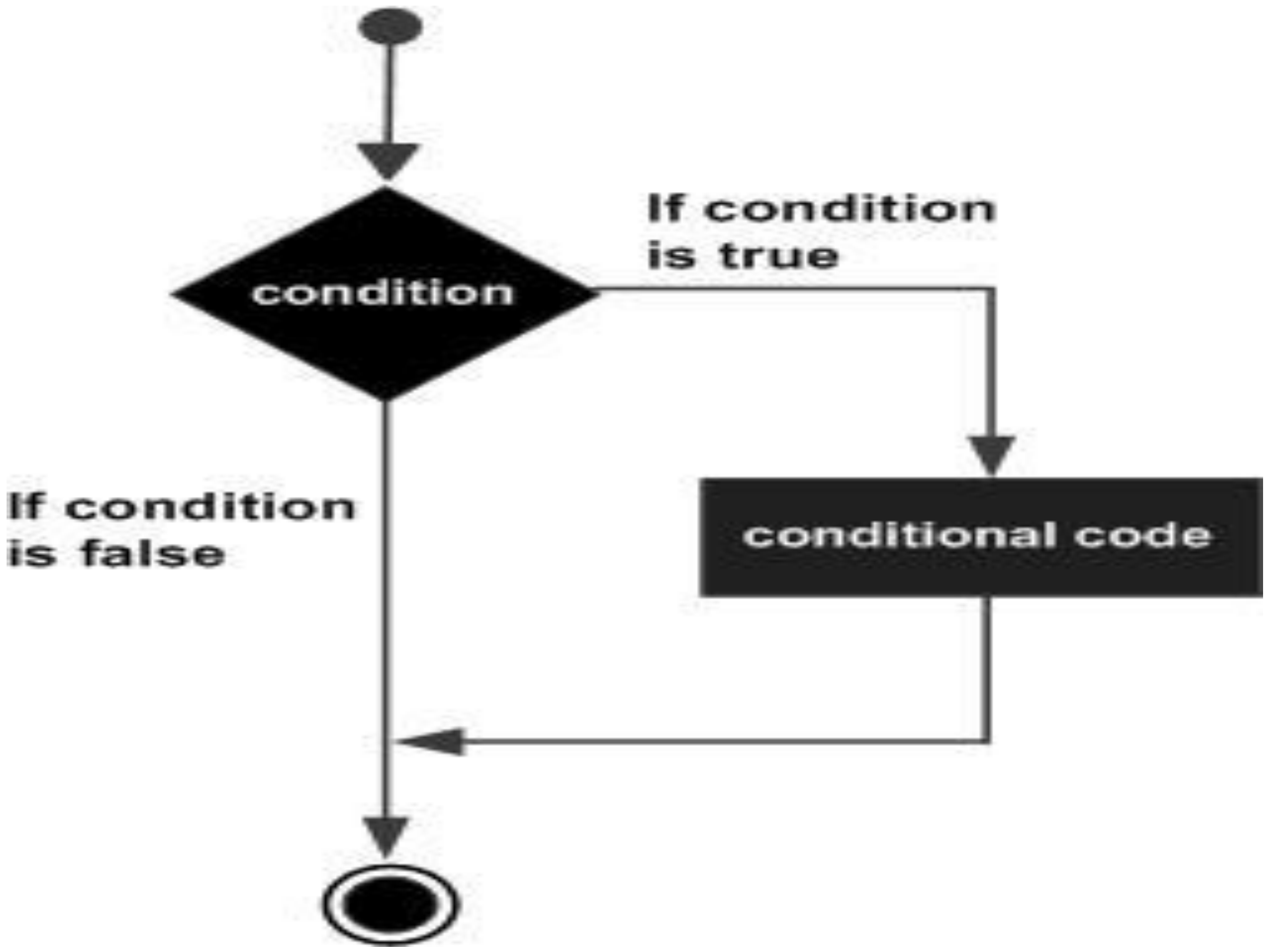
- An if statement consists of a Boolean expression followed by one or more statements.

- **Syntax :**

```
if(condition)
{
        // Statements will execute if the Boolean expression is true
}
```

- **Work flow :**

  → If the Boolean expression evaluates to true, then the block of code inside the if statement will be executed.



- **Example :**

Output :

This is if statement

```
class Test
     {
          public static void main(String [] args)
          {
               int x = 10;
               if( x < 20 )
               {
                    System.out.print("This is if statement");
               }
          }
     }
```

If else statement :

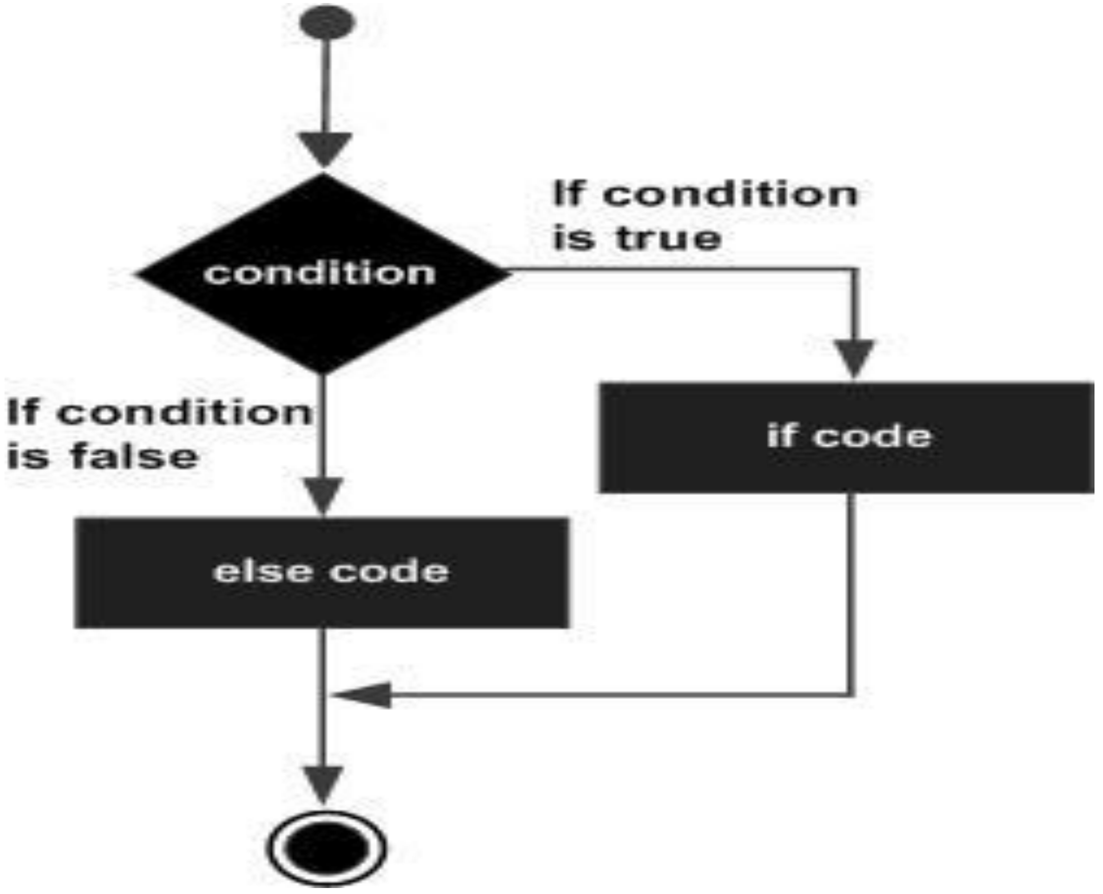
- An **if** else statement consists of a if , followed by an **else** statement.
- **Syntax :**

```
if(condition)
{
        // Statements will execute if the Boolean expression is true
}
else
{
        // Statements will execute if the Boolean expression is false
}
```

- **Work flow :**

→ if the condition is true, then the instructions inside if block will get executed.

→ if the condition is false, then the instructions inside else block will get executed.



- **Example :**

Output :

This is else statement

```
class Test
      {
            public static void main(String args[])
            {
                  int x = 30;
                  if( x < 20 )
                  {
                        System.out.print("This is if statement");
                  }
                  else
                  {
                        System.out.print("This is else statement");
                  }
            }
      }
```

If else – if statement :

- It consists of an if statement followed by *else if...else* statement, which is very useful to test various conditions.

- **Syntax :**

```
if(condition1)
{
        // Executes when the Boolean expression 1 is true
}
else if(condition2)
{
        // Executes when the Boolean expression 2 is true
}
else if(condition3)
{
\       // Executes when the Boolean expression 3 is true
}
else
{
        // Executes when the none of the above condition is true.
}
```

- **Example :**

```
class Test
         {
                  public static void main(String args[])
                  {
                           int x = 30;
                           if( x == 10 )
                           {
                           System.out.print("Value of X is 10");
                           }
                           else if( x == 20 )
                           {
                           System.out.print("Value of X is 20");
                           }
                           else if( x == 30 )
                           { System.out.print("Value of X is 30");
                           }
                           else
                           { System.out.print("This is else statement");
                           }
                  }
}
```

Switch statement :

- A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.
- **Syntax :**

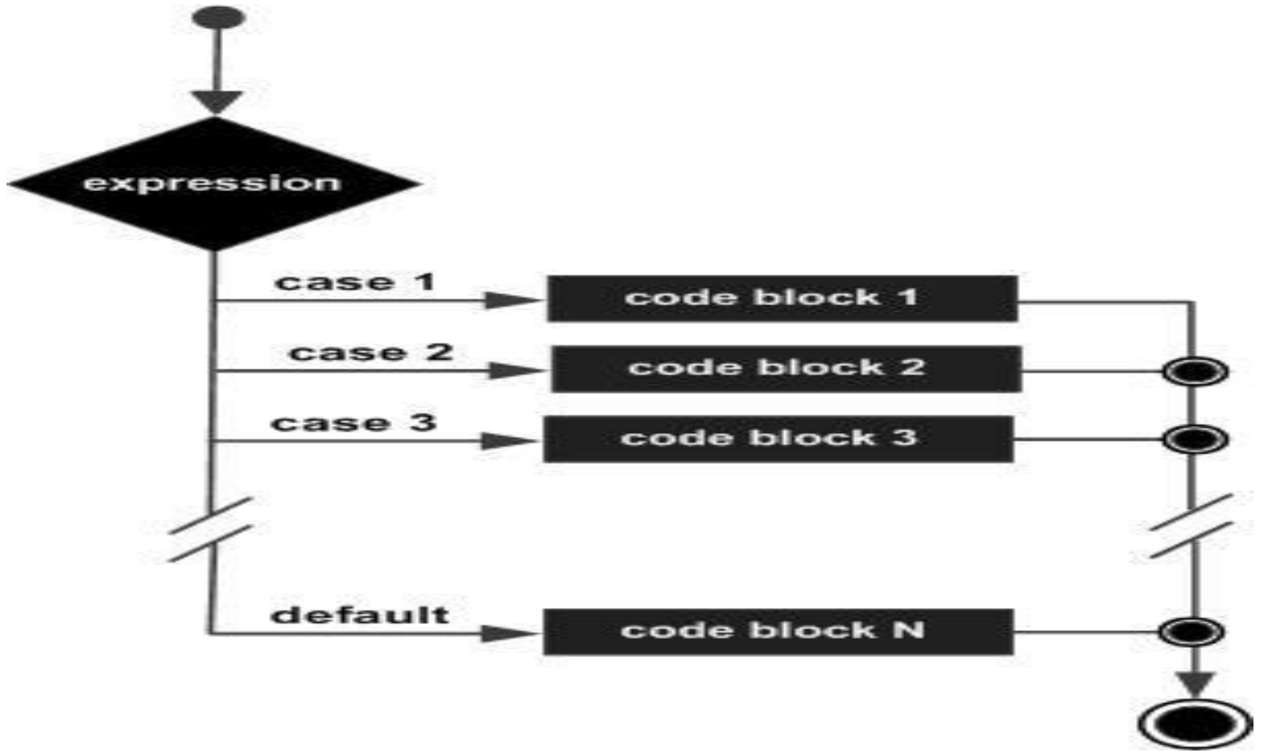
```
switch(expression)
{
        case value :
                // Statements;
        // optional case value :
                // Statements;
        .
        .
        .
                // You can have any number of case statements.
        default :
                // Statements
}
```

- **Work flow :**

→ the value /var /expression passed in the switch gets compared with the value passed in the case from top to bottom order.

→ if any of the case is satisfied, the case block will get executed and all the blocks present below gets executed.

→ if no case is satisfied, the default block gets executed.



The following rules apply to a switch statement −

- You can have any number of case statements within a switch.
- The value for a case must be the same data type as the variable in the switch .
- for a switch, we cannot pass long, float, double, boolean.
- For a case, we cannot pass variables.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.

- **Break statement :**

  → it is a keyword.

  → it is a control transfer statement.

  → when the break statement is executed, control is transferred outside the block.

Syntax :

```
switch(expression)
        {
                case value :
                        // Statements;
                break;
                // optional case value :
                        // Statements;
                break;
                .
                .
                . // You can have any number of case statements.

                default :
                        // Statements
        }
```

- **Example :**

```
class Test
        {
                public static void main(String args[])
                {
                        char grade = 'C';
                        switch(grade)
                        {
                                case 'A' :
                                                System.out.println("Excellent!");
                                break;
                                case 'B' :
                                                System.out.println("Well done");
                                break;
                                case 'C' :
                                                System.out.println("Good");
                                break;
                                case 'D' :
                                                System.out.println("You passed");
                                break;
                                case 'F' :
                                                System.out.println("Better try again");
                                break;
                                default : System.out.println("Invalid grade");
                        }
                }
}
```

LOOPING STATEMENTS

- Looping is a feature which facilitates the execution of a set of instructions repeatedly while some condition evaluates to true.

- Java provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition checking time.

  →While loop
  →Do while loop
  →For loop

While loop :

- **While loop** is a control flow statement that allows code to be executed repeatedly based on a given Boolean condition. The while loop can be thought of as a repeating if statement.

Syntax:

```
while (test_expression)
{
        // statements



        update_expression;
}
```

The various **parts of the While loop** are:

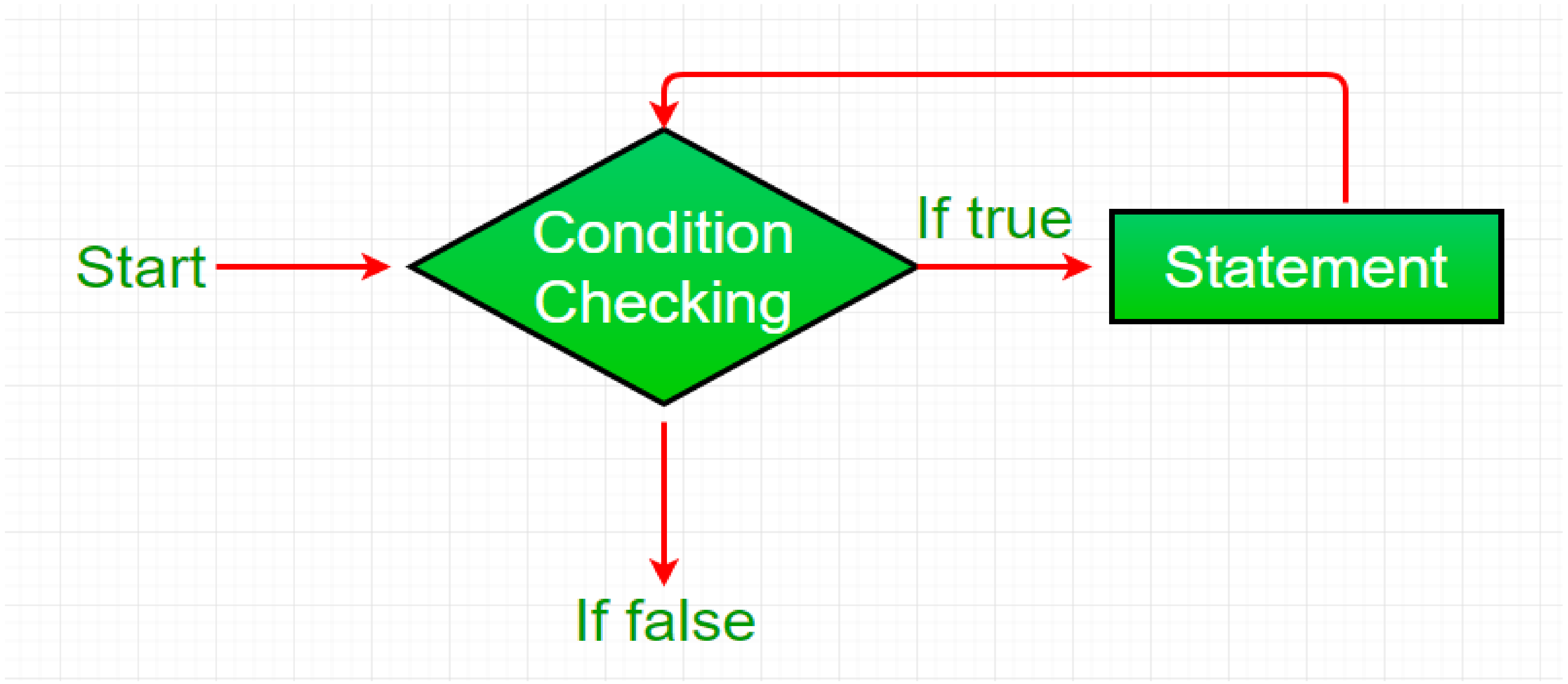
1.Test Expression: In this expression we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression.
Otherwise, we will exit from the while loop.

Example: i <= 10

2.Update Expression: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example: i++;

Flow chart while loop (for Control Flow):



- **Example : this program will print numbers from 1 to 10**

```
class whileLoopDemo
{
    public static void main(String args[])
    {
        // initialization expression
        int i = 1;

        // test expression
        while (i < =10)
            {
            System.out.println(i);

            // update expression
            i++;
        }
    }
}
```

Do while loop :

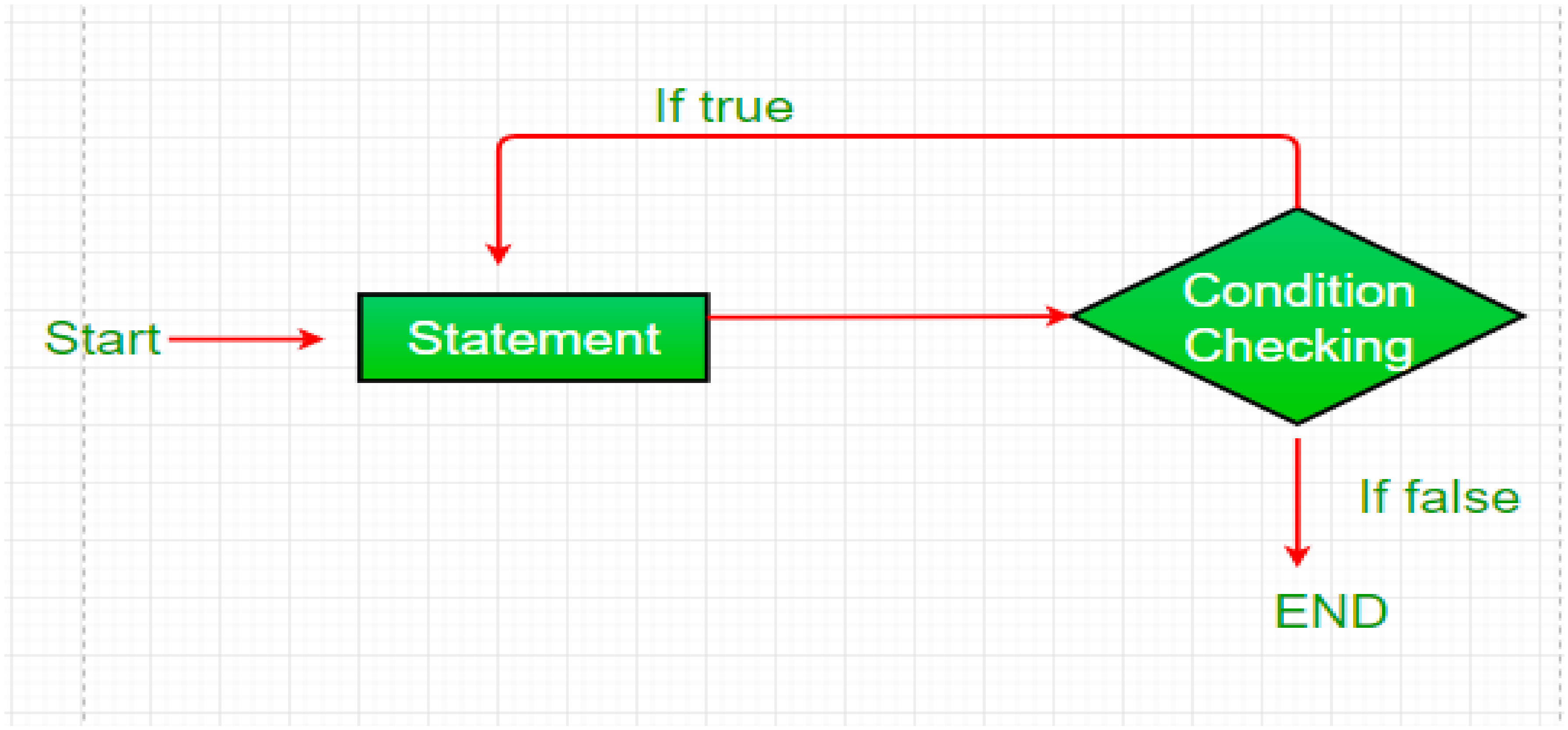
- **do-while loop** is an **Exit control loop**. Therefore, unlike for or while loop, a do-while check for the condition after executing the statements or the loop body.

Syntax:

```
do
     {
          // statements

          update_expression ;
     }
     while (test_expression);
```

Flowchart do-while loop:



- **Example : this program will print numbers from 1 to 10**

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

        // initialisation expression
        int i = 1;
        do {

            // Print the statement
            System.out.println(i);

            // update expression
            i++;
        }
        // test expression
        while (i <=10);
    }
}
```

Difference between while and do while loop:

while	do-while
• Condition is checked first then statement(s) is executed.	• Statement(s) is executed at least once, thereafter condition is checked.
• It might occur statement(s) is executed zero times, If condition is false.	• At least once the statement(s) is executed.
• If there is a single statement, brackets are not required.	• Brackets are always required.
• while loop is entry controlled loop.	• do-while loop is exit controlled loop.

For loop :

- **for loop** provides a concise way of writing the loop structure. The for statement consumes the initialization, condition and increment/decrement in one line thereby providing a shorter, easy to debug structure of looping.

Syntax:

```
for (initialization expr; test expr; update exp)
{
        // body of the loop
        // statements we want to execute
}
```

The various parts of the For loop are:

1. **Initialization Expression:** In this expression, we have to initialize the loop counter to some value.

Example: `int i=1;`

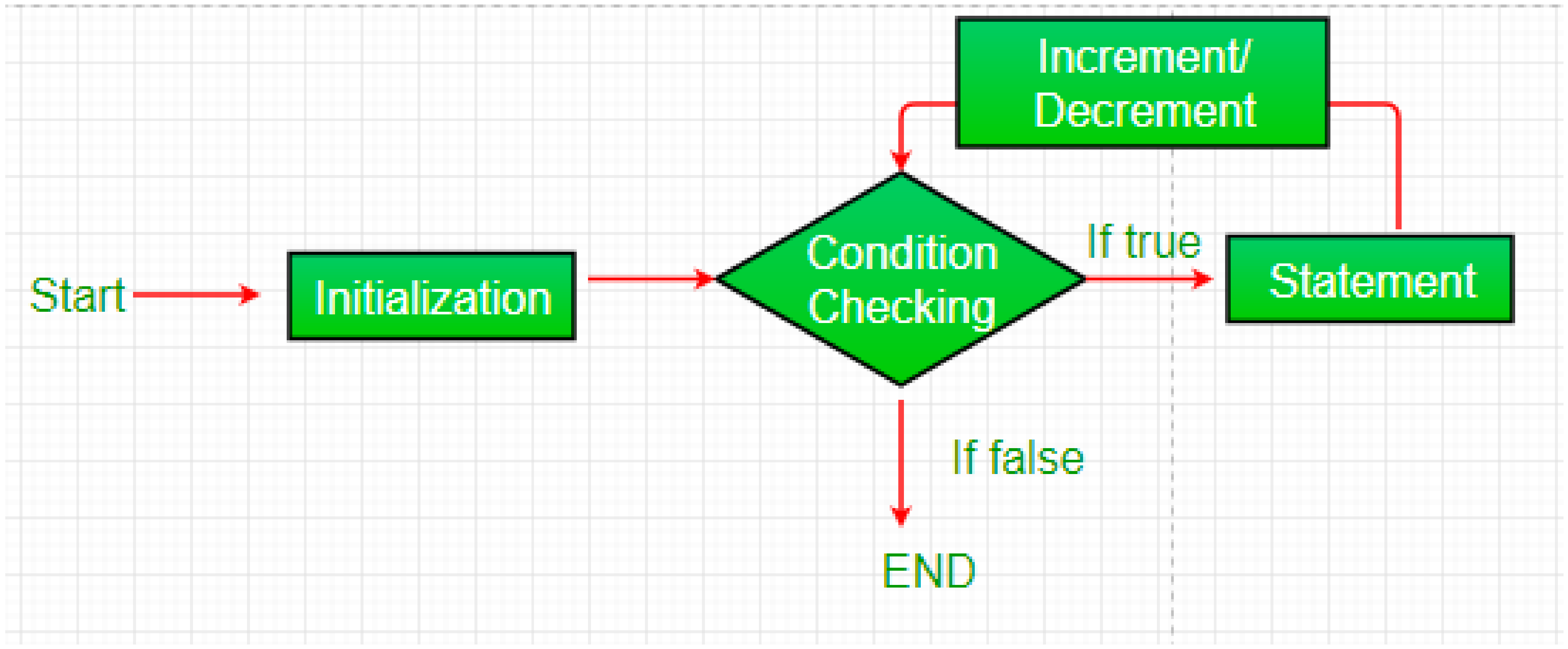
2. **Test Expression:** In this expression, we have to test the condition. If the condition evaluates to true then, we will execute the body of the loop and go to update expression. Otherwise, we will exit from the for loop.

Example: `i <= 10`

3. **Update Expression**: After executing the loop body, this expression increments/decrements the loop variable by some value.

Example: `i++;`

Flow chart for loop (For Control Flow):



- **Example : this program will print numbers from 1 to 10**

```
class forLoopDemo
{
    public static void main(String args[])
    {
        // Writing a for loop

          for (int i = 1; i <= 5; i++)
          {
                    System.out.println(i);
          }
    }
}
```

STATIC MEMBERS

- Static is a keyword.
- It is a modifier.
- Any member of a class is prefixed with static modifier, then it is known as static member of a class.
- Static members are also known as class members.
- Static members are :

  → static method

  → static variable

  → static initializers

- **Note** : static members can be prefixed only for a class member(ie.. Members declared in a class.

STATIC METHOD

- A method prefixed with the static modifier is known as static method

- **Characteristics :**
  - → static method block is stored in method area and reference is stored in class static area.
  - → We can use static methods with or without creating objects of the class.
  - → We can use the static methods with the help of class name as well as with the help of object reference.
  - A static method of the class can be used in any class with the help of class name.

- Example :

```
class Demo
{
        public static void test()
        {
                System.out.println(“hello”);
        }
        public static void main(String[  ]args)
        {
                test();
                Demo.test();
        }
}
```

Output :

```
hello
hello
```

- **Within a static context,**
  - → static members can be called directly with its name.
  - → non static members cannot be directly called with its name.
  - → this keyword cannot be used in here.

- **Static context :**
  - The block which belongs to the static method and multi line static initializer is known as static context.

STATIC VARIABLE

- Variables declared in a class block and prefixed with the static modifier is known as static variable.

- Characteristics :
    - → it is a member of a class.
    - → it will b assigned with default values.
    - → memory will be allocated in class static area.
    - → It is global in nature, it can be used within the class as well as outside the class.
    - → We can use the static methods with the help of class name as well as with the help of object reference.
    - → static variables of the class can be used in any class with the help of class name.

- **Note :** If static variables and local variables are in same name, then we can differentiate static variables with the help of class name.

- **Example :**

```
class Demo
{
        static int a;
        static int b;
        public static void main(String [ ] args)
        {
                System.out.println(a);
                System.out.println(b);
                int a=10;
                System.out.println(a);
                System.out.println(Demo.a);

        }
```

Output :

```
0
0
10
0
```

STATIC INITIALIZERS

- **We have two types:**
    - → single line static initializers
    - → multi line static initializers

- **Single line static initializers :**
    - Syntax: static datatype variable = value;
    - Example :   static int a=10;

- **Multi line static initializers :**
    - Syntax :

```
static
      {
            // statements;
      }
```

Example :

```
static
      {
            System.out.println("hii");
      }
```

- Purpose of static initializers :

→Static initializers are used to execute the start up instructions.
→The static initializers gets executed before the actual execution of main method.

- Example :

```
class Demo
{
        static
        {
                System.out.println(“ Executing first“);
        }
        public static void main(String [ ] args)
        {
                System.out.println(“executing after initializer”);
        }
```

Output :

```
Executing first
Executing after initializer
```

CLASS LOADING PROCESS

- To execute a java program, a portion of memory in RAM is allocated for JRE(Java Runtime Environment).
- In that portion of memory allocated, we have different range of memory, hence they are classified into four types :

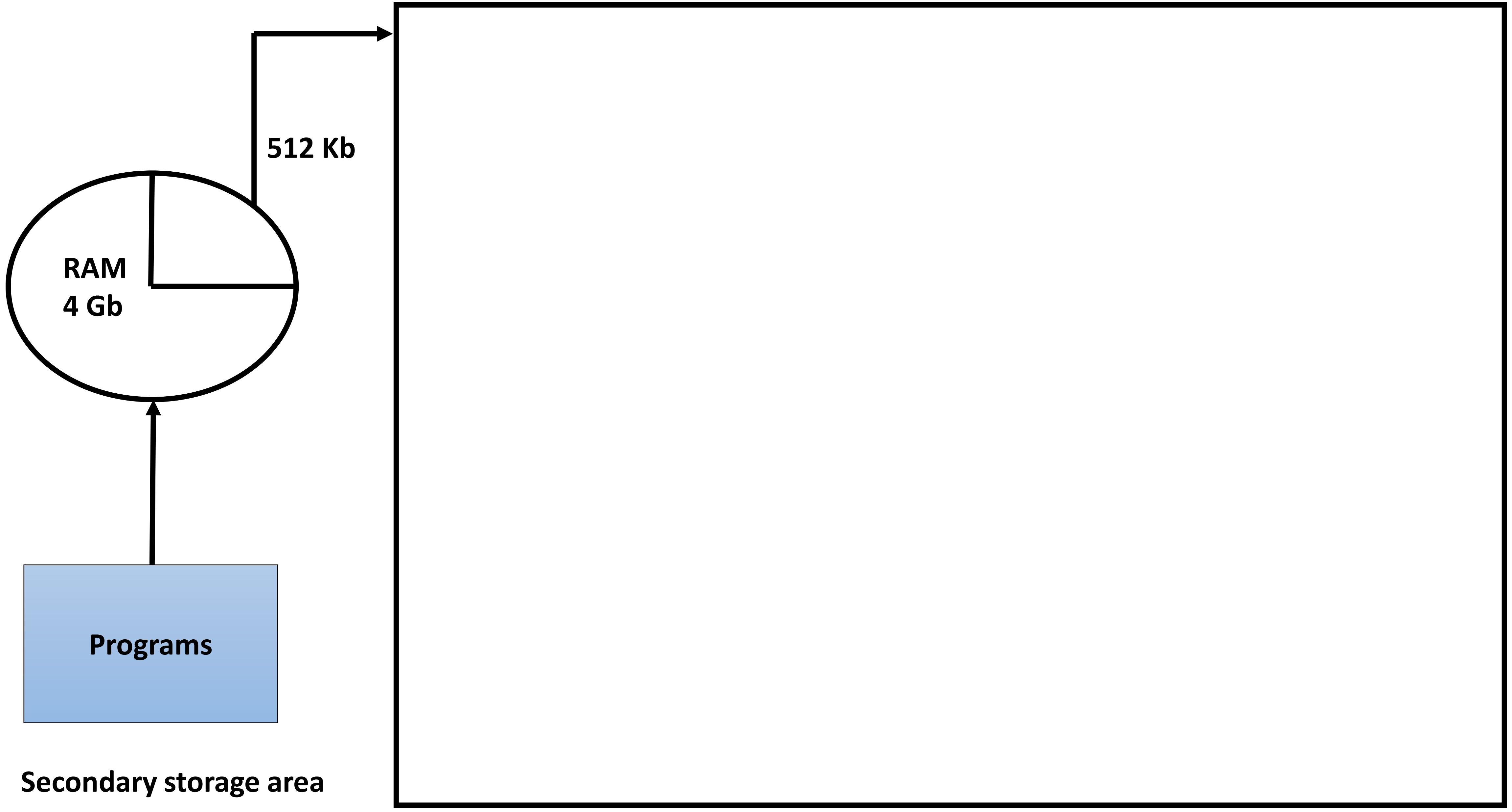
→ method area

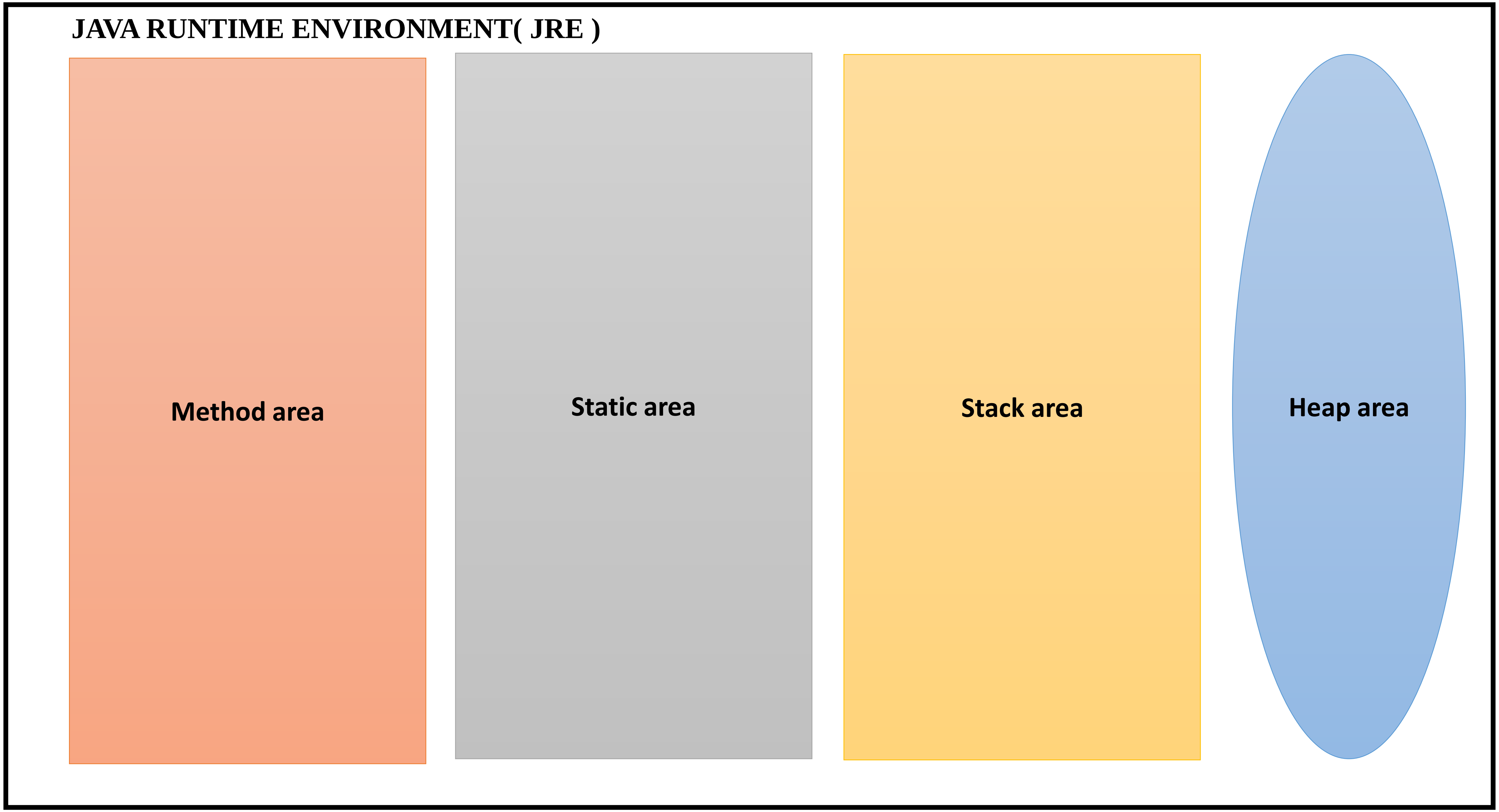
→ static area

→ stack area

→ heap area

JAVA RUNTIME ENVIRONMENT(JRE)




JAVA RUNTIME ENVIRONMENT(JRE)
Method area
Static area
Stack area
Heap area

- Method area – All the methods blocks and static multi line initializers will be stored in the method area.
- Class static area – for every class, a block of memory will be created in static area which is known as class static area.

The static members of the class will be allocated inside the memory created for the class.

- Stack area – used for execution of instructions.

For every method and multi line initializer that is under execution, a block of memory will be created in the stack area which is known as frames.

- Heap area – in heap area, a block of memory is created for object.

Every object can be accessed of a reference.

All the non static members of the class will be loaded inside block of memory in heap area.

- A block of memory is created in static area for the class which is loaded, which is known as class static area.
- All the methods (method definition) and multi line initializers will be loaded in method area and reference value will be assigned to each method block.
- All the static methods and static multi line initializers references are stored in class static area along with the method signature in the form of table.
- All the static variables are loaded into class static area and assigned with default values.
- All initializers are executed in top to bottom order.
- For every multi line static initializer, a frame is created in stack area. Once the execution is completed, the frame is removed.
- Once all the initializers are executed, we can say class loading process is complete.
- JVM will then call the main method for execution.
- A frame will be created in stack area for main method and execution of instructions begin line by line.
- For every method called, execution of main method is paused and a frame will be created in stack area for the method called.
- Once the execution of called method is completed, the frame is removed and control comes back to calling method(caller).
- The process continues. Once all instructions in main method are executed, the main method frame is removed.
- We can say execution is complete.

OBJECTS

Objects :

Any thing which has existence in real world is known as an object.

Every object will have attributes and behaviour.

objects in java :

-- According to OOPs, object is a block of memory created in heap area during runtime. It represents the real world object.

-- An object consists of attributes and behaviour. Attributes are represented with the help of non static variables. Behaviour is represented with the help of non static methods.

CLASS :

-- Before creating an object, the blueprint of the object must be designed , which provides the specification of the object. This is done with the help of class.

-- A class is a user defined non primitive data type. It represents the blueprint of the real world object.

-- A class provides specification of real world objects.

Note : We can create any number of objects for a class. Objects are also called as instance of a class.

STEPS TO CREATE AN OBJECT :

-- create a class or use an existing class which is already created.

-- Instantiation

INSTANTIATION :

-- the process of creating an object is known as instantiation.

Syntax to create an object :

Classname variablename = new Classname( );

new :

-- new is a keyword.

-- It is an unary operator.

-- It is used to create a block of memory in heap area during runtime and call the constructor.

-- Once the object is created, the new keyword will return the reference of that object.

CONSTRUCTOR :

-- It is a special member of the class whose name is same as the class name.

-- Constructor is used to load all the non static members of class into the object created.

Example :

```
class Employee
{
        String ename;
        int eid;
        String branch;
        public static void main(String [ ] args)
        {
                Employee e = new Employee ( );
                System.out.println(e.ename);
                System.out.println(e.eid);
                System.out.println(e.branch);
        }
}
```

NON STATIC MEMBERS

- Non static variables
- Non static methods
- Non static initializers
- Constructors

NON STATIC VARIABLES :

- A variable declared inside a class block and not prefixed with a static modifier is known as non static variable.

- **CHARACTERISTICS :**

- We cannot use the non static variable without creating an object.
- We can only use the non static variable with the help of object reference.
- Non static variables are assigned with default values during the object loading process.
- Multiple copies of non static variables will be created (Once for each object).

NON STATIC METHODS

- A method declared inside a class block and not prefixed with a static modifier is known as non static method.

- **CHARACTERISTICS :**

- All non static methods will get stored inside the method area and reference of these non static methods will be stored inside the object created in heap area.
- We cannot call the non static methods without creating an object of the class.
- We cannot access the non static methods directly with the class name.

Non static context :

- The block which belongs to the non static method and multi line non static initializer is known as non static context.
- Inside a non static context, we can use static and non static members of the class directly by using its name.
- Inside a static context, Non static members cannot be accessed directly with their names .

NON STATIC INITIALIZER

- It will execute during the object loading process.
- It will execute once for every object of a class created.

- **Purpose of non static initializers :**

- Non static initializers are used to execute the start up instructions for an object.

- **Types of non static initializers :**

- Single line non static initializer
- Multi line non static initializer

- **Single line static initializers :**
    - Syntax: datatype variable = value;
    - Example : int a=10;
- **Multi line static initializers :**
    - Syntax :

```
{
        // statements;
}
```

Example :

```
{
        System.out.println(“hii”);
}
```

- **Example** :

```
class Demo
{
          {
                    System.out.println(“ Executing first“);
          }

          {
                     System.out.println(“ Executing second”);
          }
          public static void main(String [ ] args)
          {
                    System.out.println(“executing main method”);
                    Demo d = new Demo ();
          }
}
```

Output :

```
Executing main method
Executing first
Executing second
```

OBJECT LOADING PROCESS EXAMPLE :

EXAMPLE :

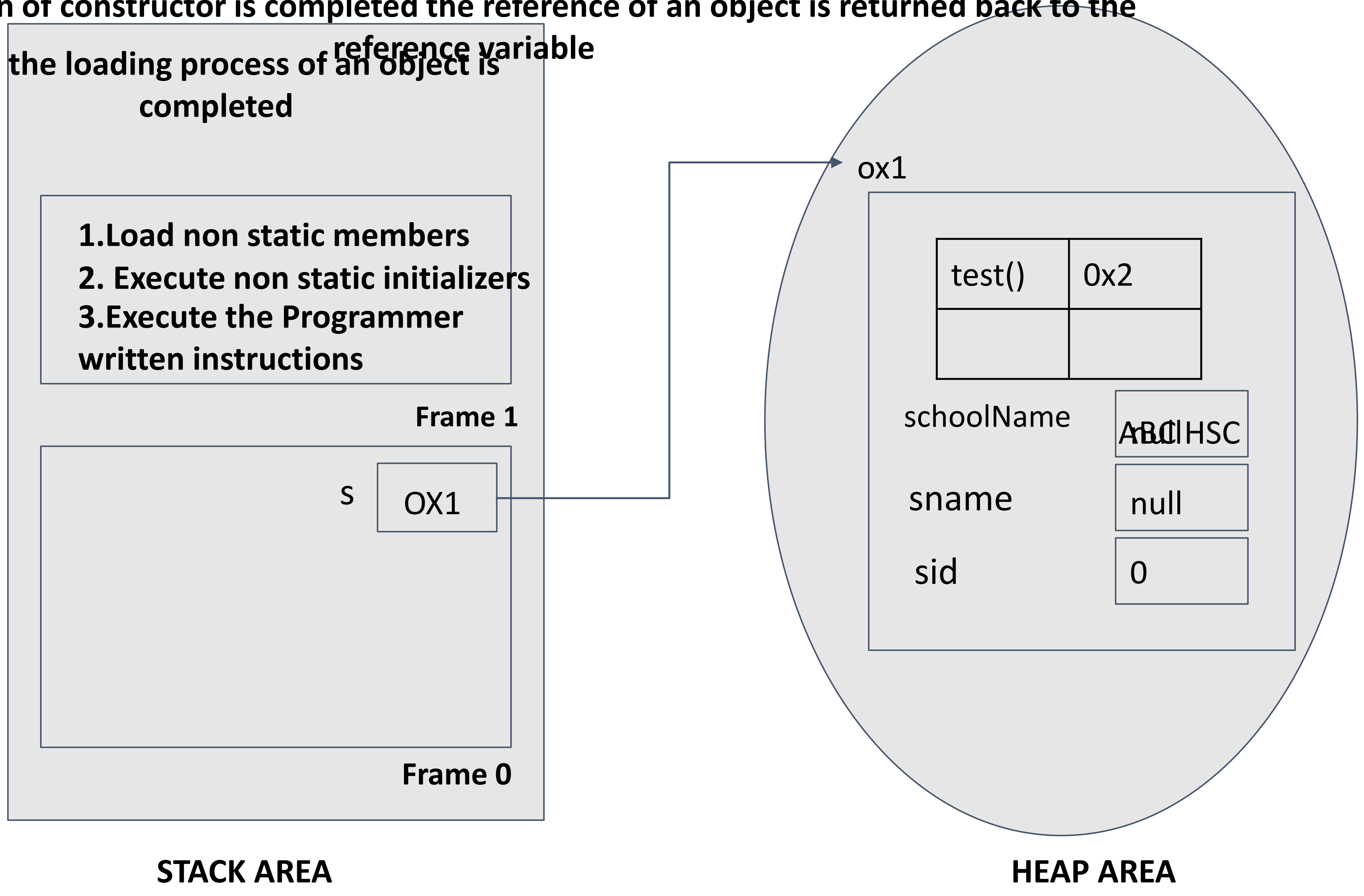
```
class School
{
        String schoolName = “ABC HSC”;
        {
                S.o.pln(“School name is “+ schoolName);
        }
        public void test()
        {
                S.o.pln(“From test”);
        }
        String sname;
        Int sid;
        public static void main(String []args)
        {
            School s = new School();
        }
}
```

New keyword will create a block of memory in a heap area.

Constructor is called

Once execution of constructor is completed the reference of an object is returned back to the reference variable

Thus the loading process of an object is completed



1.Load non static members
2. Execute non static initializers
3.Execute the Programmer written instructions

THIS KEYWORD :

- It is a keyword.
- It is a non static variable which holds the reference of the currently executing object.

- **Uses of this keyword :**
- It is used to access the members of current object.
- It is used to give the reference of the current object.
- Reference of a current object can be passed down from a method using this keyword.
- Calling a constructor of the same class is achieved with the help of this keyword.

<u>CONSTRUCTOR</u>

Constructor :

- A constructor is a special type of non static method whose name is same as the class name but it does not have a return type.

- Syntax :

```
[ access modifier ] [ modifier ] classname ( [ formal arguments ] )
{
    // Initialization
}
```

CONSTRUCTOR BODY :

- A constructor body will have following things :
  - → Load all the non static members into the object created.
  - → Non static initializers of the class are executed.
  - → Programmer written instructions are executed.

PURPOSE OF CONSTRUCTOR :

- During the execution of constructor,
  - →Non static members of the class will be loaded into the object created.
  - →If there is a non static initializers in the class, they are executed from top to bottom order.
  - →Programmer written instructions of the constructor gets executed.

Note : If the programmer fails to create a constructor, then the compiler will add a default no argument constructor.

CLASSIFICATION OF CONSTRUCTOR :

Constructors can be classified into 2 types based on formal arguments,

1. No argument constructor : A constructor which doesn’t have a formal argument is known as no argument constructor.

2. Parameterized argument : A constructor which consists of formal arguments is known as parameterized constructor.

NO ARGUMENT CONSTRUCTOR :

Syntax :

```
[ access modifier ] [ modifier ] classname ( )
{
        // instructions;
}
```

Note : If the programmer fails to create a constructor, compiler will add a no argument constructor implicitly.

• **Example :**

```
class Student
{
        String name;
        int id;
        Student  ( )
        {
                System.out.println(“ no argument constructor “):
        }
        public static void main(String [ ]  args)
        {
                Student s = new Student ( );
                s.name = “ abc “;
                s.id = 10;
                System.out.println( “name : “+name+” id : “+id );
        }
}
```

Output :

```
abc
10
```

PARAMETERIZED CONSTRUCTOR :

- The constructor which has formal arguments is called as parameterized constructor.
- The purpose of parameterized constructor : They are used to initialize the non static variables by accepting the data from the constructor in the object loading process.

Syntax :

```
[ access modifier ] [ modifier ] classname ( formal arguments )
{
        // Initialization ;
}
```

- **Example :**

```
class Student
{
        String name;
        int id;
        Student  (String name, int id )
        {
                this.name=name;
                this.id=id;
        }
        public static void main(String [ ]  args)
        {
                Student s = new Student ( “ Ram “, 10);
                System.out.println(s.name);
                System.out.println(s.id);
        }
}
```

Output :

```
Ram
10
```

LOADING PROCESS OF AN OBJECT

- New keyword will create a block of memory in heap area.
- It then calls the constructor.
- During the execution of constructor,

    → All non static members of the class are loaded into the object.

    → If there are non static initializers, they are executed from top to bottom order.

    → Programmer written instructions of the constructor will be executed.

- The execution of the constructor is completed.
- The object is created successfully.
- The reference of the object created is returned by the new keyword.
- These steps are repeated for every object creation.

CONSTRUCTOR OVERLOADING

- If a class is having more than one constructor , it is known as constructor overloading.
- Rule : The signature of constructors must be different.

<u>Example :</u>

```
class Student
{
        String sname;
        int sid;
        Student ( )
        {
        }
        Student(String sname)
        {
                this.sname=sname;
        }
        Student(int sid)
        {
                this.sid=sid;
        }
        Student(String sname, int sid)
        {
                this.sname=sname;
                this.sid=sid;
        }
}
```

Creating objects for Student class :

Student s = new Student (“Seeta “ , 4);

Student s = new Student (“Seeta “);

Student s = new Student (4);

Student s = new Student ( );

CONSTRUCTOR CHAINING

- A constructor calling another constructor is known as constructor chaining.

- In java, we can achieve constructor chaining by using 2 ways :

→ this ( )

→ super ( )

this ( ) statement :

- It is used to call the constructor from another constructor within the same class.
- **Rule :**

1. this ( ) can be used only inside the constructor.
2. It should always be the first statement in the constructor.
3. The recursive call to the constructor is not allowed.
4. If a class has n constructors, it can have n-1 this ( ) statements.

NOTE : If a constructor has this ( ) statement , then the compiler doesn't add load instructions and non static initializers into the constructor body.

Example :

```
class Student
{
            String sname;
            int sid;
            long cno;
            Student ( )
            {
            }
            Student(String sname)
            {
                        this.sname=sname;
            }
            Student(String sname, int sid )
            {
                        this(sname);
                        this.sid=sid;
            }
            Student(String sname, int sid, long cno)
            {
                        this(sname, sid);
                        this.cno=cno;
            }
}
```

Creating an object for Student class :

Student s = new Student (“Seeta “ , 4 , 123456789);

<u>PRINCIPLES OF OOPS</u>

OOPs has following principles :

1. **ENCAPSULATION**
2. **INHERITANCE**
3. **POLYMORPHISM**
4. **ABSTRACTION**

ENCAPSULATION

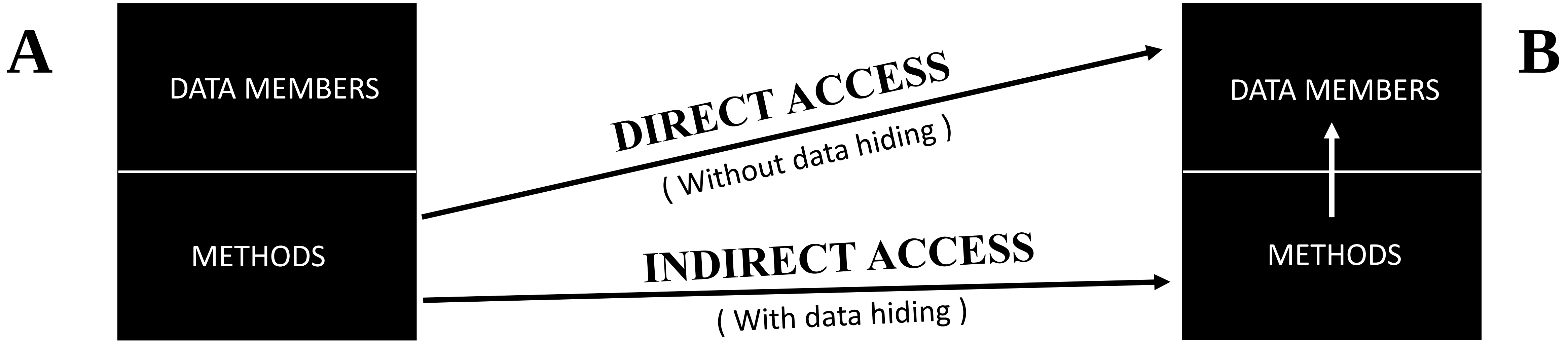
- **Encapsulation** **:** The process of binding the state (Attributes) and behaviours of an object together is known as encapsulation.

- We can achieve encapsulation in java with the help of class. A class has both state and behaviour of an object.

- **Advantage** : By using encapsulation, we can achieve data hiding.

- **Data Hiding** : It is a process of restricting the direct access of data members of an object and providing indirect secured access of data members via methods (public) of same object is known as data hiding.

- Data hiding helps to achieve verification and validation of data before storing and modifying it.



- With data hiding, direct access is not possible, only indirect access is allowed.
- Without data hiding, direct access is possible, also indirect access is possible.

- **Steps to achieve data hiding :**

1. Prefix the data members of the class with private modifier.
2. Design a getter and setter method.

PRIVATE MODIFIER :

- private is an access modifier.
- It is a class level modifier.
- If the members of the class are prefixed with private modifier, then we can access those members only within the class.

NOTE : Data hiding can be achieved with the help of private modifier.

- **Example :**

```
class Book
{
        private int book_id;
        private String book_name;
        private double price;

        Book ( int book_id, String book_name, double price)
        {
                this.book_id=book_id;
                this.book_name=book_name;
                this.price=price;
        }
        public static void main(String[] args)
        {
                Book b=new Book(1214, " JAVA ", 450.0);
                System.out.println ( "book id : "+book_id+"book name " +book_name+" price " +price);
        }
}
```

GETTER AND SETTER METHODS

Getter method :

- It is used to fetch/get the data .
- The return type of getter method is the type of hidden value.

Setter method :

- It is used to update or set the data.
- The return type is always void.

- **NOTE :** The validation and verification can be done in setter method before storing data and in getter method before reading the private data members.

- If you want to make your hidden value only readable, then create only getter method.

- If you want to make your hidden value only modifiable, then create only setter method.

- If you want to make your hidden value both readable and modifiable , then create both getter and setter method.

- If you want to make your hidden value neither readable nor modifiable, then don’t create a getter and setter method.

<u>ADVANTAGES OF DATA HIDING :</u>

- Provides security to the data members.
- We can verify and validate the data before modifying it.
- We can make the data members of the class to
  - → only readable
  - → only modifiable
  - → both readable and modifiable
  - → neither readable nor modifiable

IS-A RELATIONSHIP

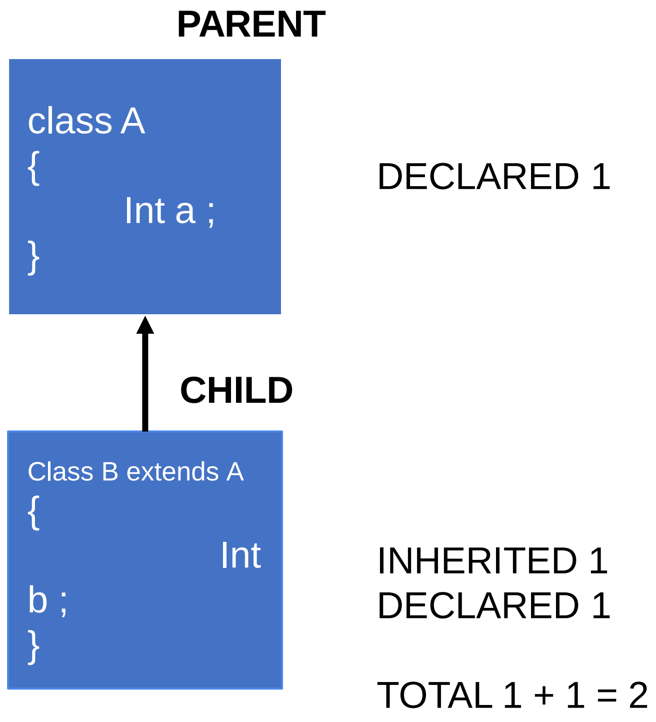
Is-A RELATIONSHIP :

- The relationship between two objects which is similar to the parent and child relation is known as the Is-A relationship.
- In an Is-A relationship, the child object will acquire all properties of the parent object, and child object will have its own extra properties.
- In an Is-A relationship, we can achieve generalization and specialization.

NOTE :

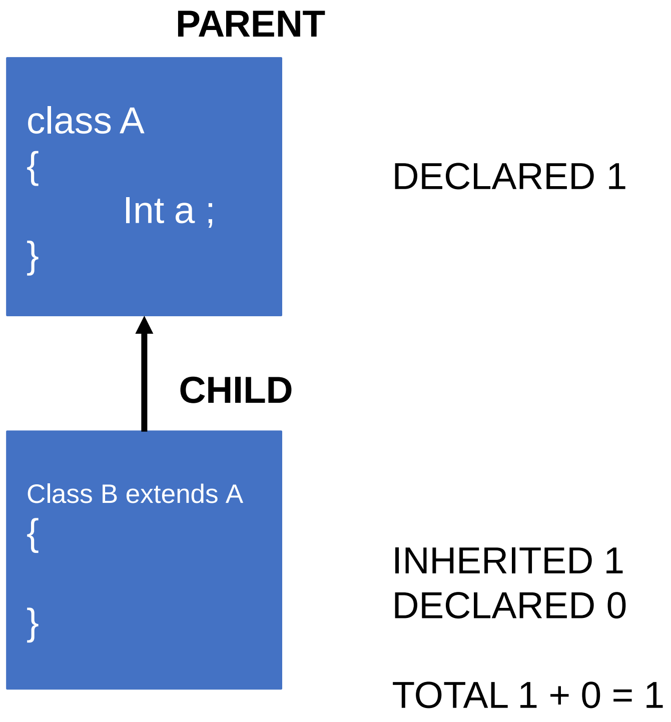
- **Parent is called generalized.**
- **Child is called specialized.**

EXAMPLE :



```
class A
{
        Int a ;
}
```

```
Class B extends A
{
                Int
b ;
}
```



```
class A
{
        Int a ;
}
```

```
Class B extends A
{

}
```

- With the help of the child class reference type, we can use the members of the parent's class as well as the child.
- With the help of parent class reference, we can use only the members of a parent but not the child class.

TERMINOLOGIES

PARENT CLASS :

The parent class is also known as a superclass or base class.

CHILD CLASS :

The child class is also known as a subclass or derived class.

NOTE : Is-A relationship is achieved with the help of inheritance.

INHERITANCE :

The process of one class acquiring all the properties and behavior from the other class is called inheritance.

In java, we can achieve inheritance with the help of

1. **extends keyword**
2. **implements keyword**

EXTENDS KEYWORD

extends keyword :

extends keyword is used to achieve inheritance between two classes.

EXAMPLE :

```
class A
{
        Int i = 10;
}
Class B extends A
{
        Int j = 20;
        public static void main(String argos[])
        {
                B b = new B();
                S.o.pln( b.i ); //CTS
                S.o.pln( b.j ); //CTS
                A a = new A();
                S.o.pln( a.i );  //CTS
                S.o.pln( a.i )  //CTE with the help of parent variable we can only access parent members
        }
}
```

SESSION 1 ACTIVITY

EXAMPLE PROGRAM 1

Step1: Create a class A.

Step2: Declare and initialize Non-static variables.

Step3: Create a method display() to print - Hello from A.

Step4: Create a class B and make it a child class for A.

Step5: Declare and initialize Non-static variables.

Step6: create an object for class B.

Step7: Call the display() method of class A and print the values in the variables with the reference variable of class B.

EXAMPLE PROGRAM 2

Step1: Create a class A.

Step2: Declare Non-static variables.

Step3: Create a method display() to print - Hello from A.

Step4: Create a class B and make it a child class for A.

Step5: Declare Non-static variables.

Step6: create an object for class B.

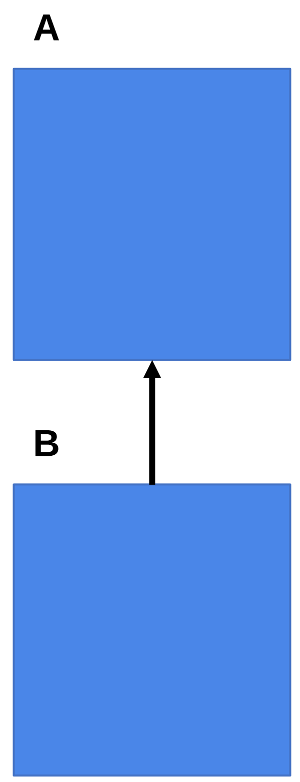
Step7: Initialize all the variables.

Step8:Call the display() method of class A and print the values in the variables with the reference variable of class B.

Types of inheritance

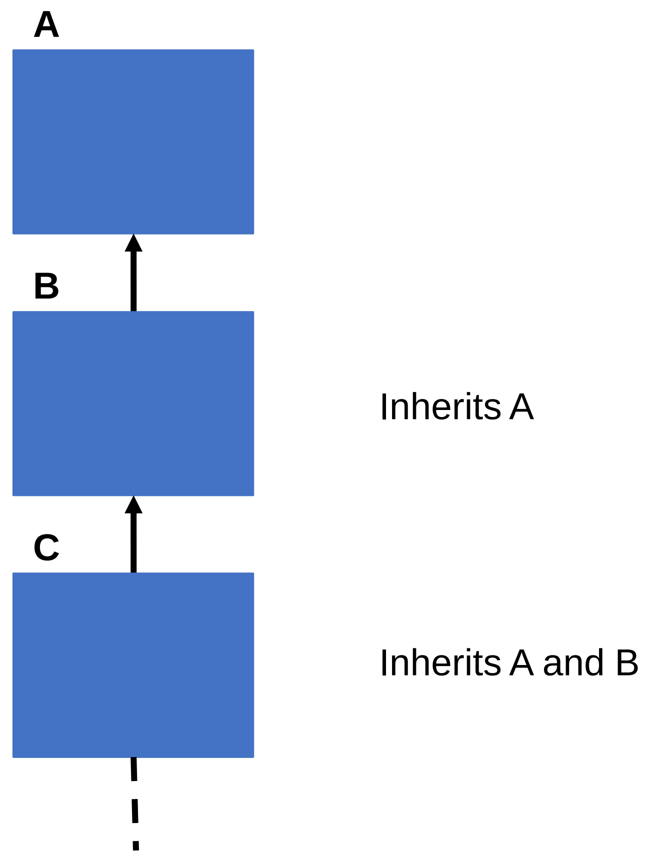
1.SINGLE LEVEL INHERITANCE :

Inheritance of only one level is known as single-level inheritance.



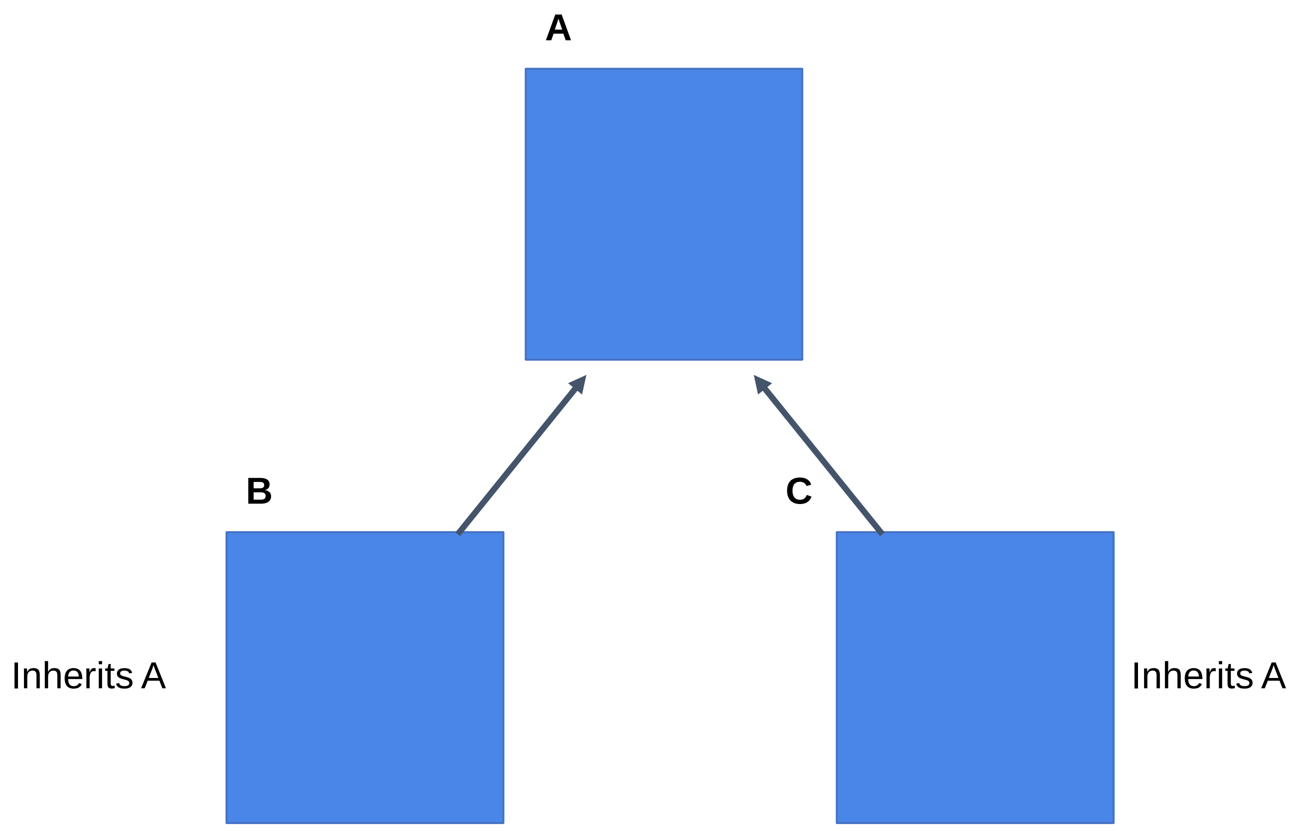
2. MULTI LEVEL INHERITANCE :

Inheritance of more than one level is known as multi-level inheritance.



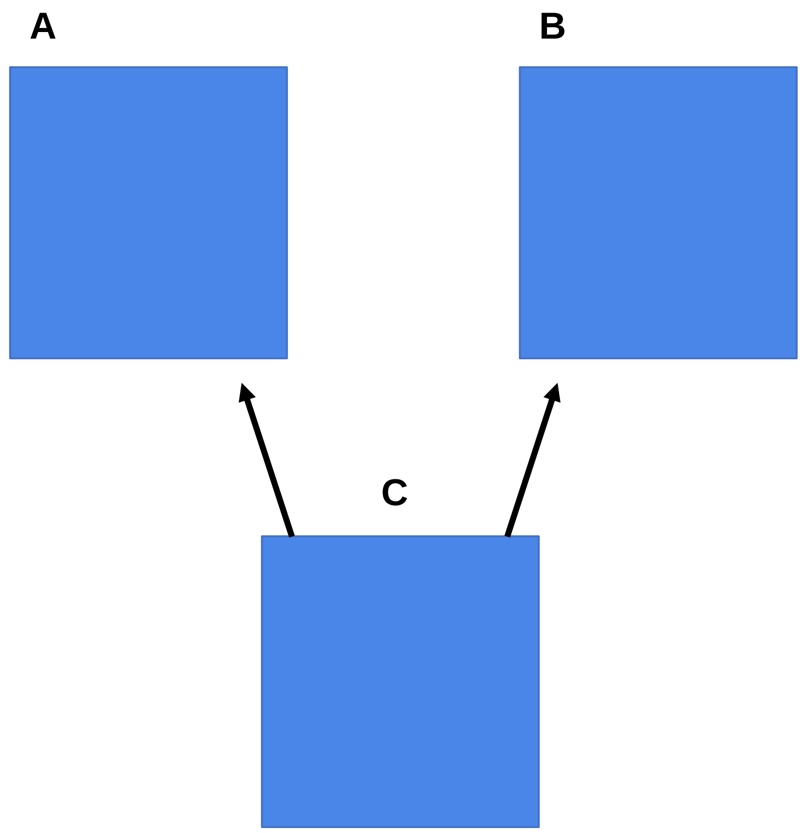
3. HIERARCHICAL INHERITANCE :

If a parent (superclass) has more than one child (subclass) at the same level then it is known as hierarchical inheritance.



MULTIPLE INHERITANCE :

If a subclass (child) has more than one parent (Super class) then it is known as multiple inheritance.



NOTE :

- Multiple inheritance has a problem known as the diamond problem.
- Because of the diamond problem, we can’t achieve multiple inheritance only with the help of class.
- In java, we can achieve multiple inheritance with the help of an interface.

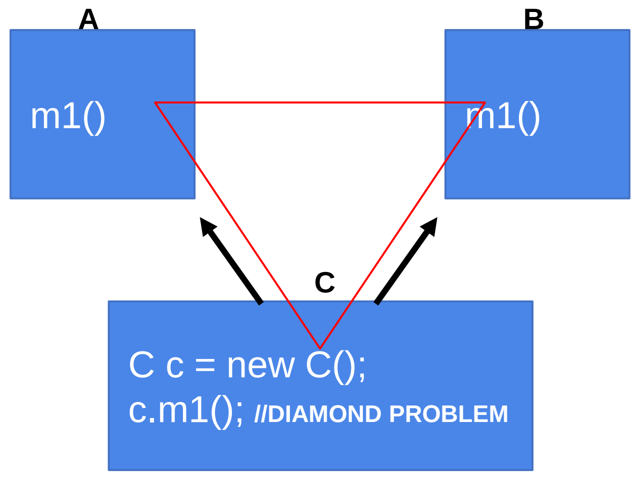
DIAMOND PROBLEM

DIAMOND PROBLEM :

Assume that there are two classes A and B. Both are having the method with the same signature. If class C inherits A and B then these two methods are inherited to C .

Now an ambiguity arises when we try to call the super class method with the the help of subclass reference.

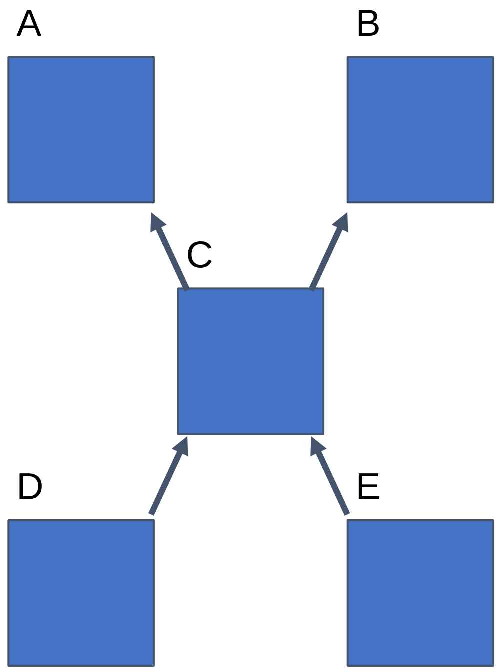
This problem is known as the **diamond problem.**

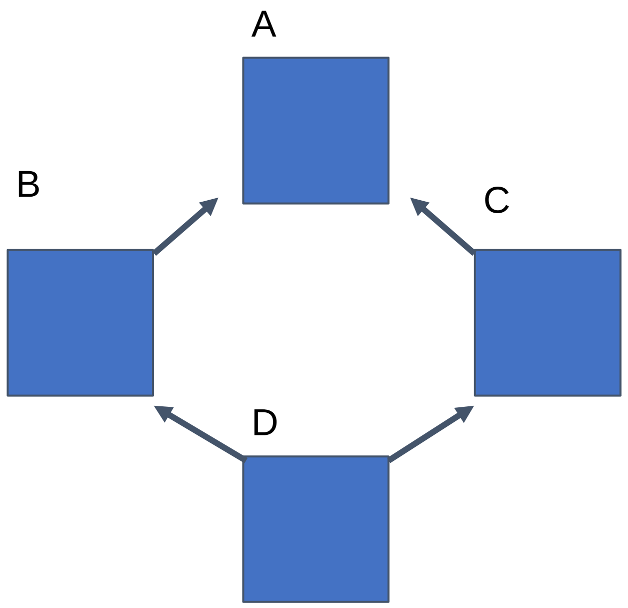


HYBRID INHERITANCE

HYBRID INHERITANCE :

The combination of multiple inheritance and hierarchical inheritance is known as hybrid inheritance.





EXAMPLE PROGRAM 1

Step1: Create a class Author.

Step2: create three Non-static variables (authorName, age and place)

Step4: Create a class Book make it a child class for Author.

Step5: create two Non-static variables (name and price).

Step6: Create a parameterized constructor and initialize (name, price and author).

Step7: Create a main method.

Step8: Create object for Book class.

Step9: Print the values present in variables (name, price, authorName, age and place).

SUPER() CALL STATEMENT

super() CALL STATEMENT :

- super is a keyword, it is used to access the members of the superclass.
- super() call statement is used to call the constructor of the parent class from the child class constructor.

PURPOSE OF SUPER() STATEMENT :

- When the object is created, the super call statement helps to load the nonstatic members of the parent class into the child object.
- We can also use the super() call statement to pass the data from the subclass to the parent class.

RULE TO USE SUPER() STATEMENT

RULE TO USE SUPER() STATEMENT

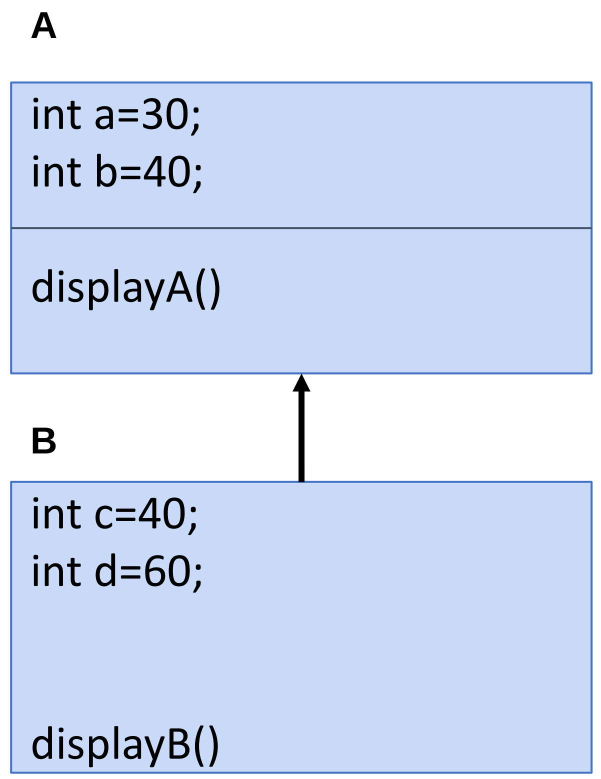
- super() call statement should always be the first instruction in the constructor call.
- If a programmer doesn't use the super() call statement, then the compiler will add a no-argument super call statement into the constructor body.

DIFFERENCE BETWEEN THIS() AND SUPER()

this()	**super()**
It is used to call the constructor of the same class	It is used to call the constructor of the parent class(Super class)
It is used to represent the instance of child class	It is used to represent the instance of parent class
It should be used as a first statement in a constructor block	It should be used as a first statement in a constructor block

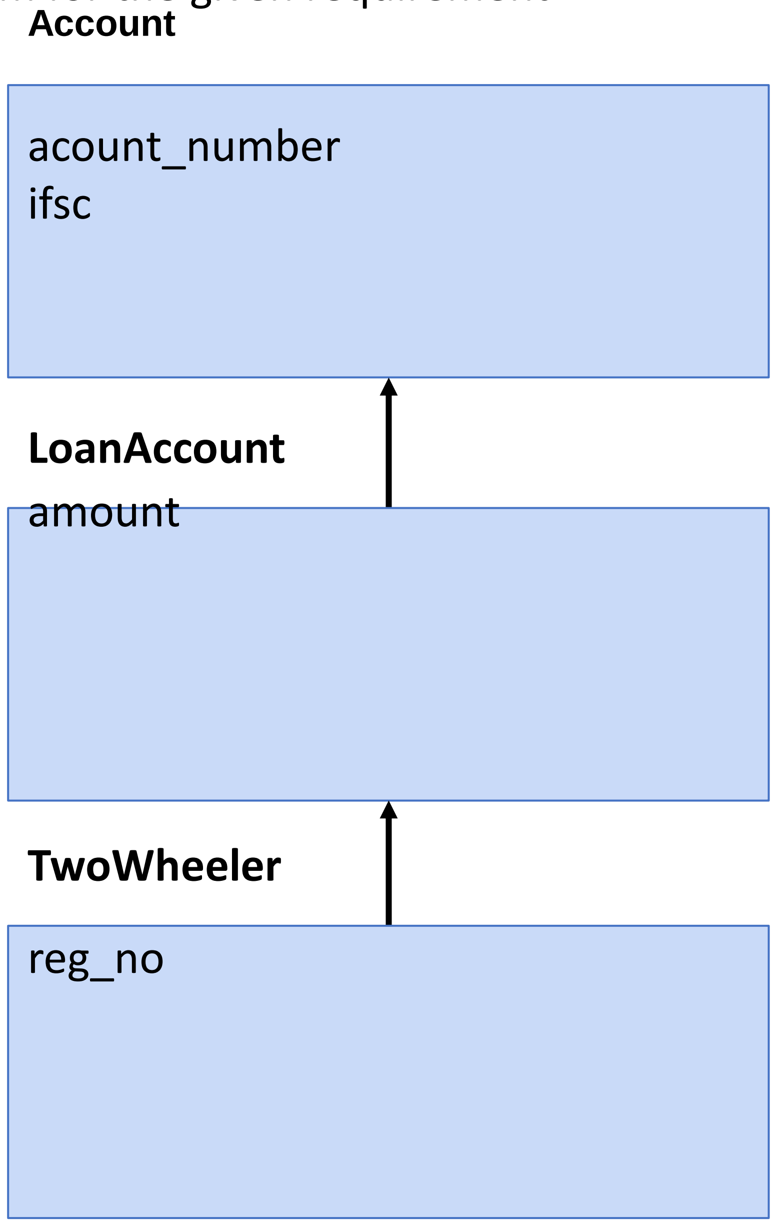
ASSIGNMENT

Write a java program for the given requirement



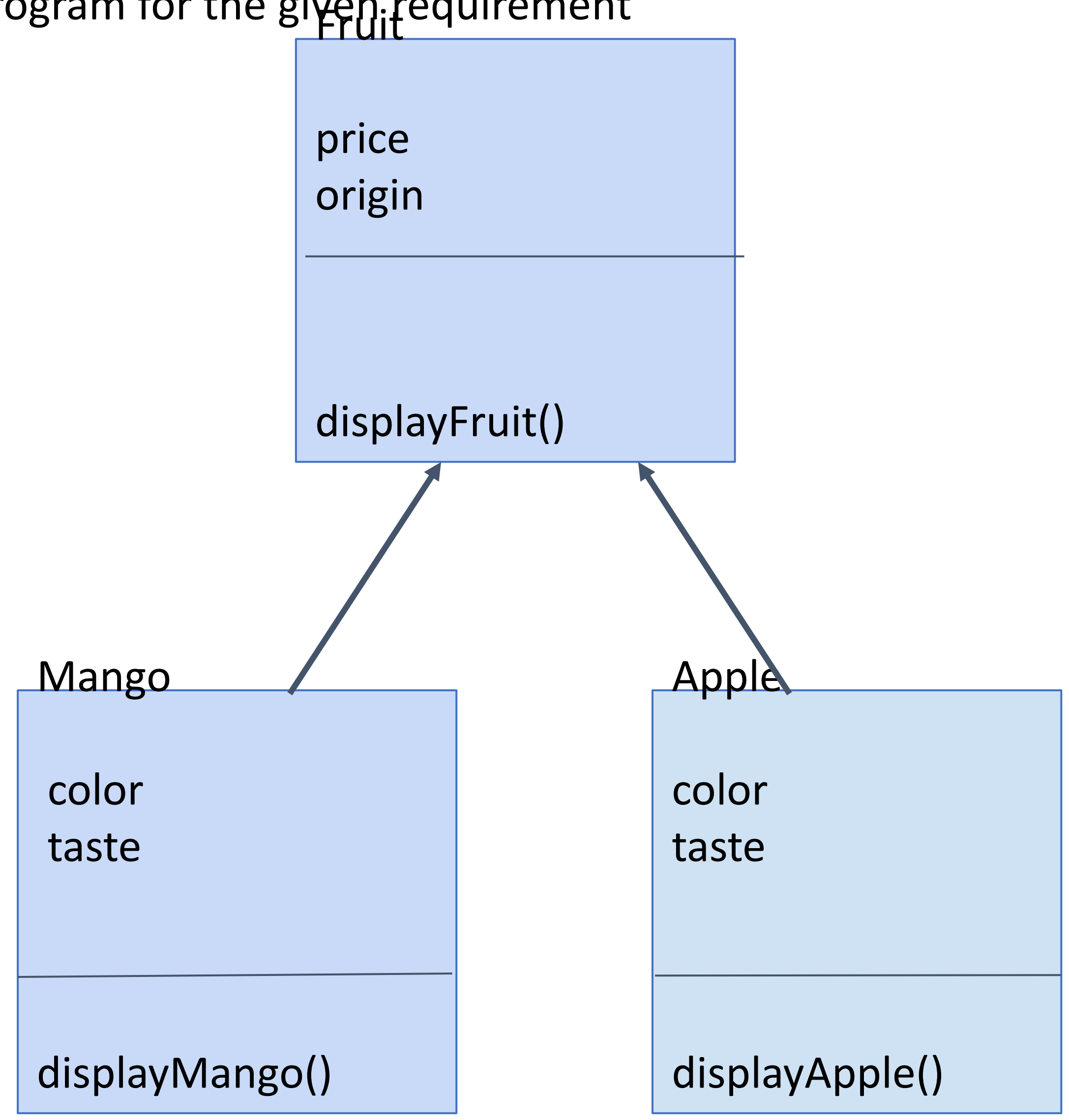
Display the values in the variables using reference variable of class B

Write a java program for the given requirement



Initialize and Display the values in the variables using reference variable of class

Write a java program for the given requirement



- Display the values in the variables using reference variable of class Mango
- Display the values in the variables using reference variable of class Apple

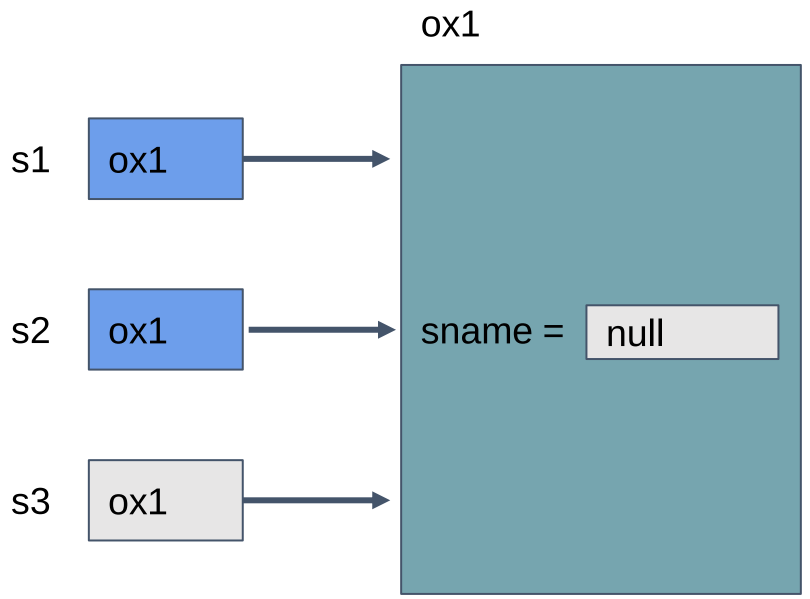
UNDERSTANDING REFERENCE VARIABLE

UNDERSTANDING REFERENCE VARIABLE :

- One object can be referred to with multiple reference variables.
- We can copy the reference of one object into multiple reference variables.

EXAMPLE :

```
class Student
{
        String sname ;
}
Student s1 = new Student();
Student s2 = s1 ;
Student s3 = s1 ;
```



NOTE :

We can access the members of the Student object by using s1 or s2 or s3. [The state of the object can be modified by using any of the reference variables.]

NOTE :

- **We can copy the reference from one variable to another variable only if both the reference variable are the same type.**
- **If both the reference variable are different types then conversion of one reference type to another is required.**

NON PRIMITIVE TYPE CASTING

NON PRIMITIVE TYPE-CASTING (DERIVED TYPE CASTING)

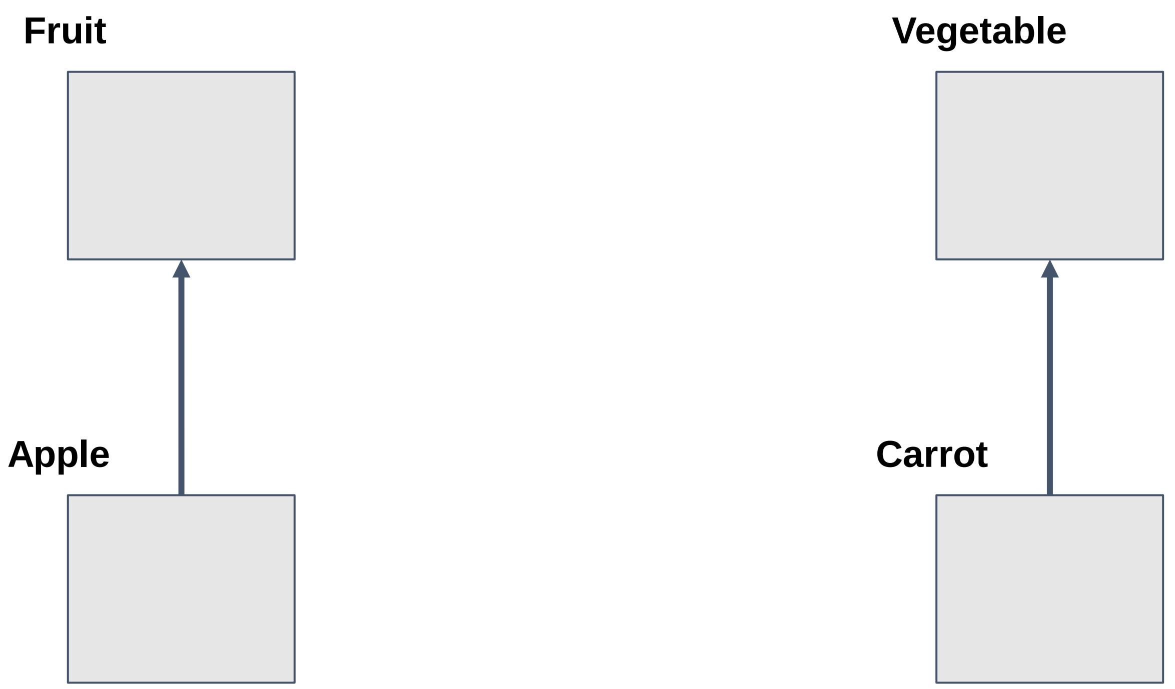
The process of converting one reference type into another reference type is known as non-primitive or derived type casting.

RULES TO ACHIEVE NON PRIMITIVE TYPE CASTING :

We can convert one reference type into another reference type only if it satisfies the following condition,

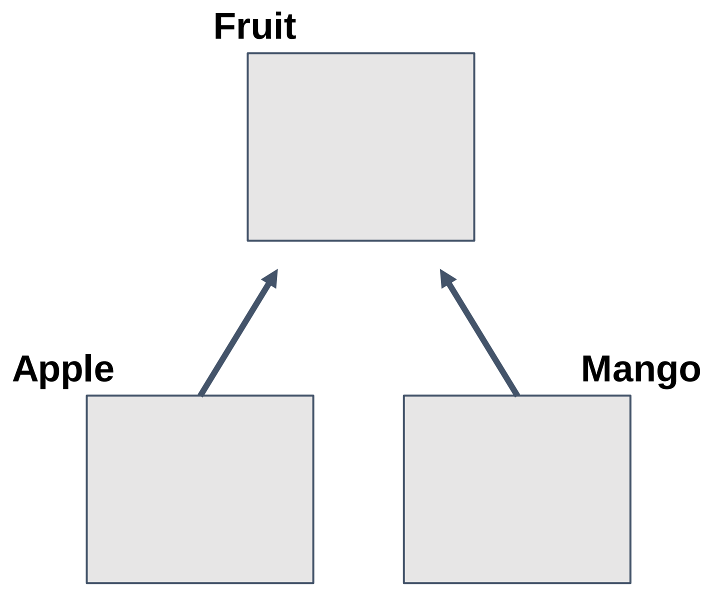
- There must be an Is-A relation (Parent and child) exist between two references.
- If the class has a common child.

EXAMPLE 1 :



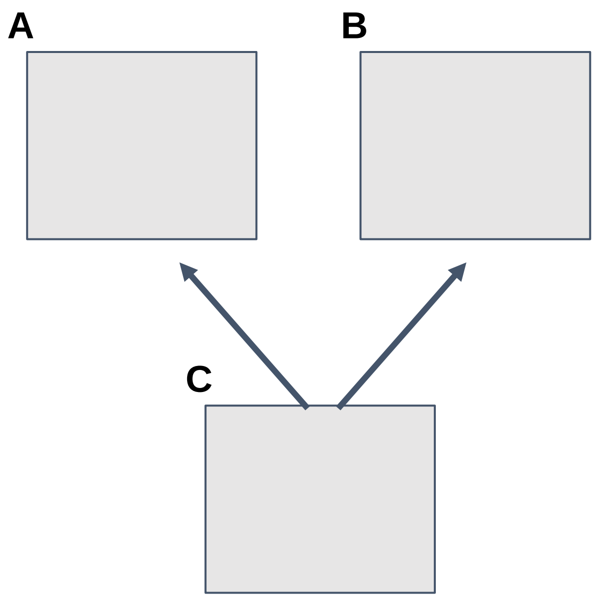
- Fruit can be converted to Apple and Apple can be converted to fruit as well as Vegetable can be converted to Carrot and Carrot can be converted to Vegetable.
- But Fruit and apple can’t be converted to Vegetable and carrot as well as Vegetable and carrot can’t be converted to Fruit and Apple.

EXAMPLE 2 :



- Fruit can be converted to Apple as well as Mango and Mango and Apple can be converted into Fruit.
- But Apple can’t be converted into Mango as well as Mango can’t be converted into Apple**.**

EXAMPLE 3 :



We can convert A to B type , B to A type , A to C type , C to A type , B to C type and C to B type.

TYPES OF DERIVED TYPE CASTING

TYPES OF NON PRIMITIVE OR DERIVED TYPE CASTING :

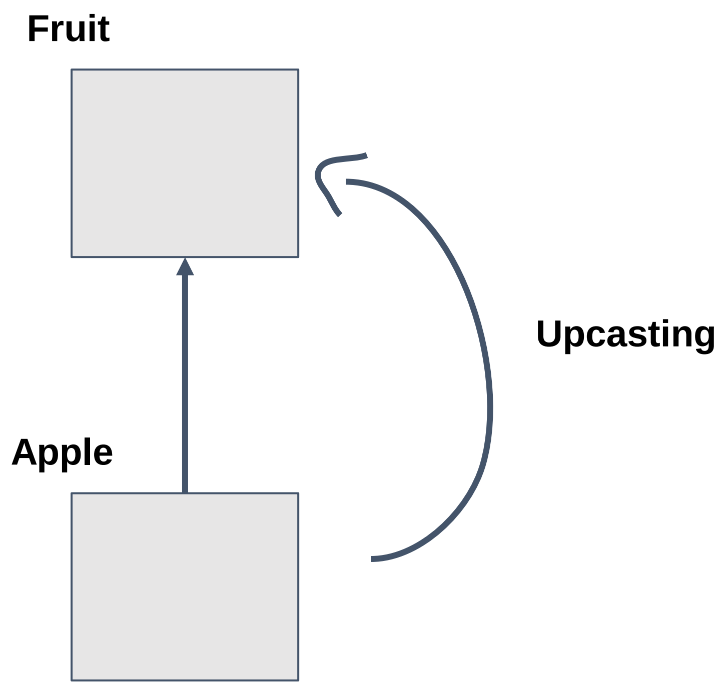
Non primitive type casting can be further classified into two types,

I. **Upcasting**
II. **Downcasting**

UPCASTING

UPCASTING :

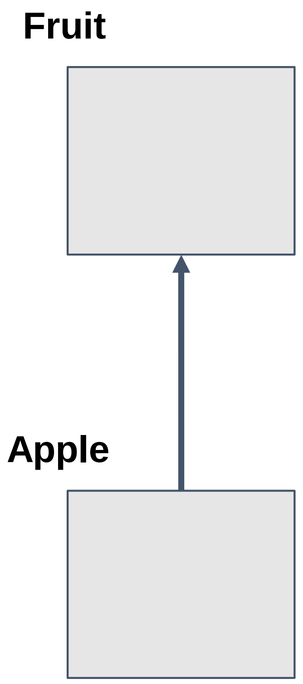
The process of converting the child class reference into a parent class reference type is known as upcasting.



NOTE :

- **The upcasting is implicitly done by the compiler.**
- **It is also known as auto upcasting.**
- **Upcasting can also be done explicitly with the help of a typecast operator.**
- **Once the reference is upcasted we can’t access the members of the child.**

EXAMPLE :



```
Apple a = new Apple();
Fruit f = a; // Upcasting
```

NOTE :

By using ‘f’ we can access only the members of aFruit (Super class)

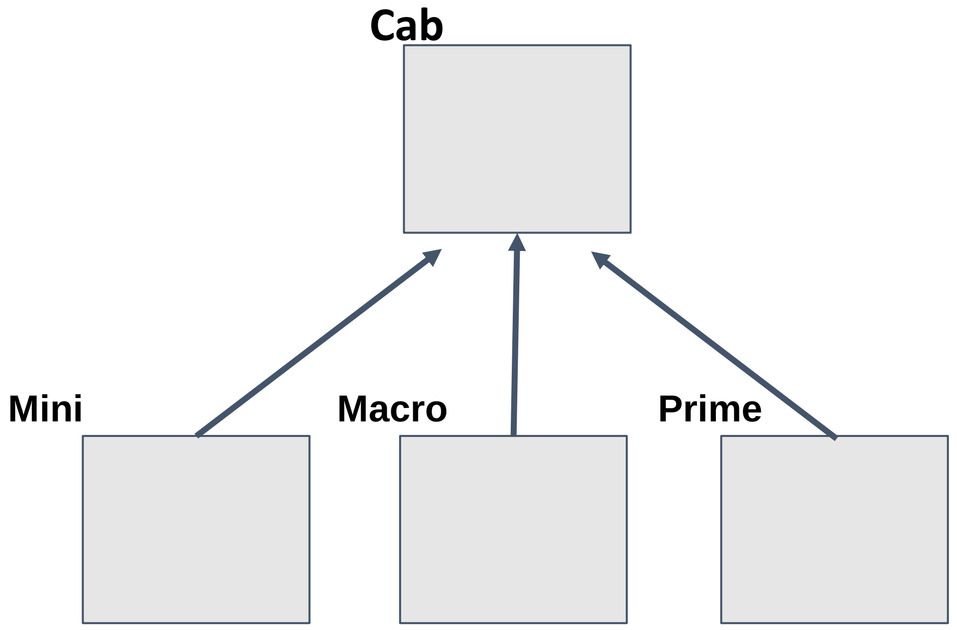
WHY DO WE NEED UPCASTING ?

- It is used to achieve generalization.
- It helps to create a generalized container so that the reference of any type of child object can be stored.

EXAMPLE :

Generalized variable

```
Cab c ;
c = new Mini();
c = new Macro();
c = new Prime();
```



DISADVANTAGE

DISADVANTAGE :

There is only one disadvantage of upcasting that is, once the reference is upcasted its child members can’t be used.

NOTE :

In order to overcome this problem, we should go for downcasting

EXAMPLE PROGRAM 1

Step1: Create a class Fruit.

Step2: create a variable fruit and initialize it.

Step3: create a class Vegetable.

Step4: create a vegetable variable and initialize it.

Step5: Create a class Driver.

Step6: create a main method.

Step7: create an object for the Class Fruit and store the reference in a variable of type Fruit.

Step8: create an object for the Class Vegetable and store the reference in a variable of type Vegetable.

Step9: create an object for the Class Vegetable and store the reference in a variable of type Fruit.

Step10: create an object for the Class Fruit and store the reference in a variable of type Vegetable.

EXAMPLE PROGRAM 2

Step1: Create a class Parent.

Step2: create a variable p and initialize it.

Step3: Create a class Child and make it a child class for Parent.

Step4: create a variable c and initialize it.

Step5: Create a class Driver.

Step6: create a main method.

Step7: create an object for the Class Child and store the reference variable of type Child and print the values present in the variables p and c.

Step8: create an object for the Class Child and store the reference variable of type Parent and print the values present in the variables p and c.

DOWNCASTING

DOWNCASTING :

The process of converting a parent (superclass) reference type to a child (subclass) reference type is known as downcasting.

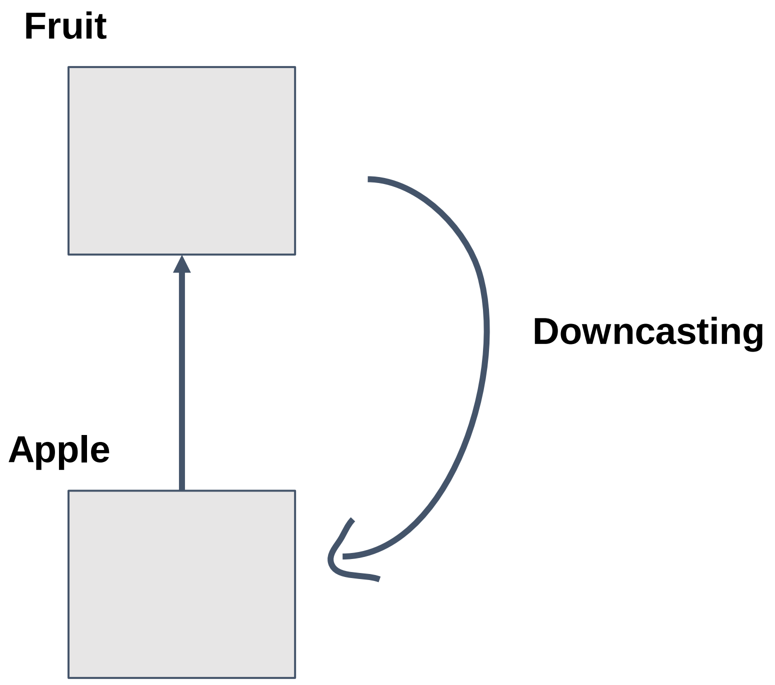
NOTE :

- **Downcasting is not implicitly done by the compiler.**
- **It should be done explicitly by the programmer with the help of a typecast operator.**

PURPOSE OF DOWNCASTING

WHY DO WE NEED A DOWNCASTING ?

- If the reference is upcasted, we can’t use the members of a subclass.
- To use the members of a subclass we need to downcast the reference to a subclass.



CLASS CAST EXCEPTION

ClassCastException :

- It is a RuntimeException.
- It is a problem that occurs during runtime while downcasting.

When and why do we get a ClassCastException?

When we try to convert a reference to a specific type(class), and the object does't have an instance of that type then we get ClassCastException.

EXAMPLE:

```
Child c = (Child)new Parent(); //ClassCastException
```

instanceof operator

instanceof operator :

- It is a binary operator
- It is used to test if an object is of given type.
- The return type of this operator is boolean.
- If the specified object is of given type then this operator will return true else it return false.

Syntax to use instanceOf operator :

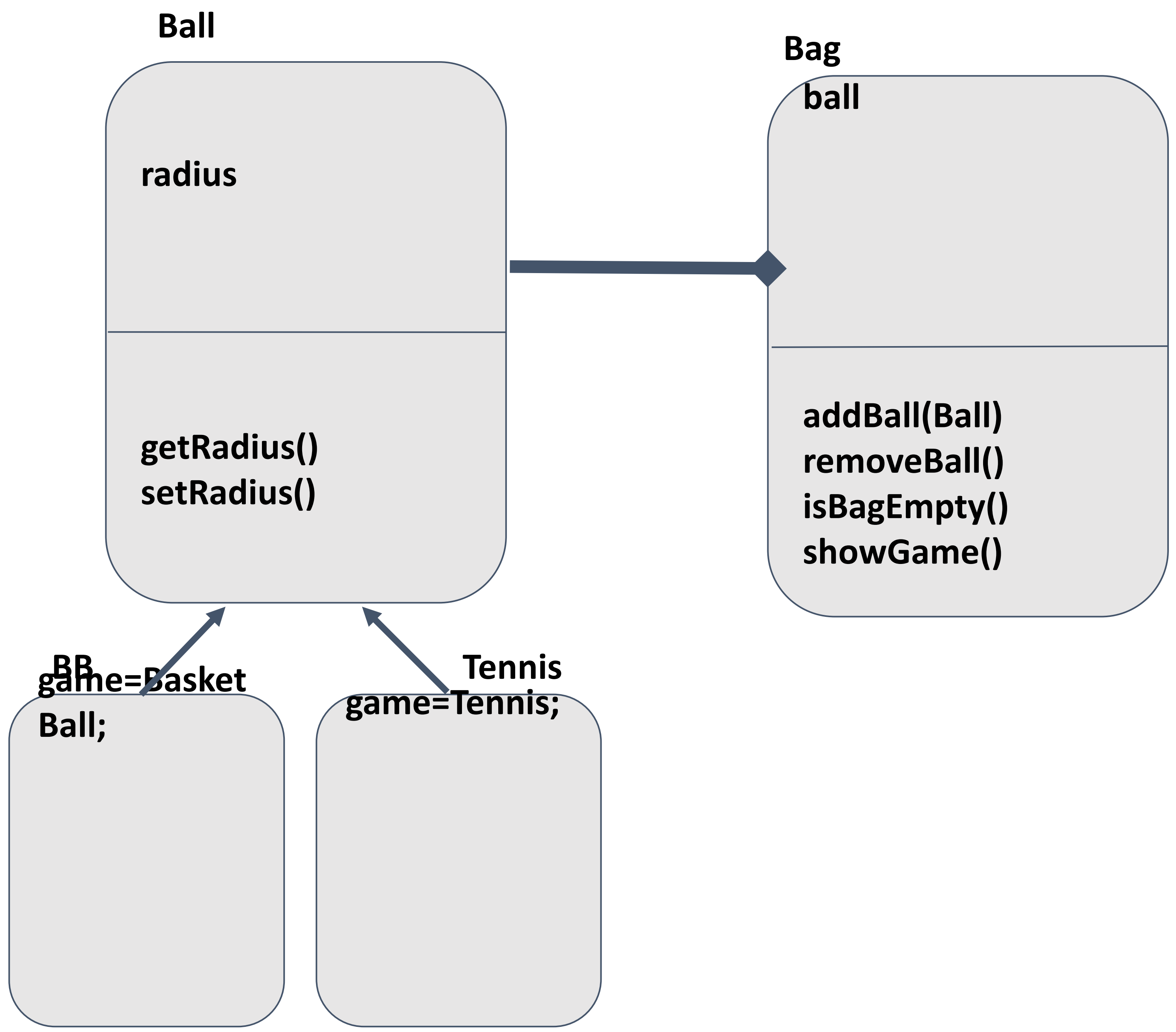
(Object_Ref) instanceof (type)

EXAMPLE : new String() instanceof Object ;

NOTE :

There must be Is-A relation exist between Object_Ref and type passed in an instanceof operator otherwise we will get a Compile time error.

PROGRAMS TO DEMONSTRATE THE USE OF UPCASTING AND DOWNCASTING



User Interface

S1 : create object of bag.

S2:Display Menu

1. Add Ball
2. Remove Ball
3. Check Bag is empty or not
4. Show game that can be played

S3:Read the choice.

POLYMORPHISM

POLYMORPHISM

Polymorphism is derived from two different Greek words 'Poly' means Numerous 'Morphs' means form Which means Numerous form. Polymorphism is the ability of an object to exhibit more than one form with the same name.

For Understanding :

One name -----> Multiple forms

One variable name ------> Different values

One method name -----> Different behaviour

TYPES OF POLYMORPHISM :

In java we have two types of polymorphism,

1. **Compile time polymorphism**
2. **Runtime Polymorphism**

COMPILE TIME POLYMORPHISM

COMPILE TIME POLYMORPHISM :

- If the binding is achieved at the compile-time and the same behavior is executed it is known as compile-time polymorphism.
- It is also said to be static polymorphism.
- It is achieved by

1. **Method overloading**
2. **constructor overloading**
3. **Variable shadowing**
4. **Method shadowing**
5. **Operator overloading (does not supports in java)**

METHOD OVERLOADING

METHOD OVERLOADING :

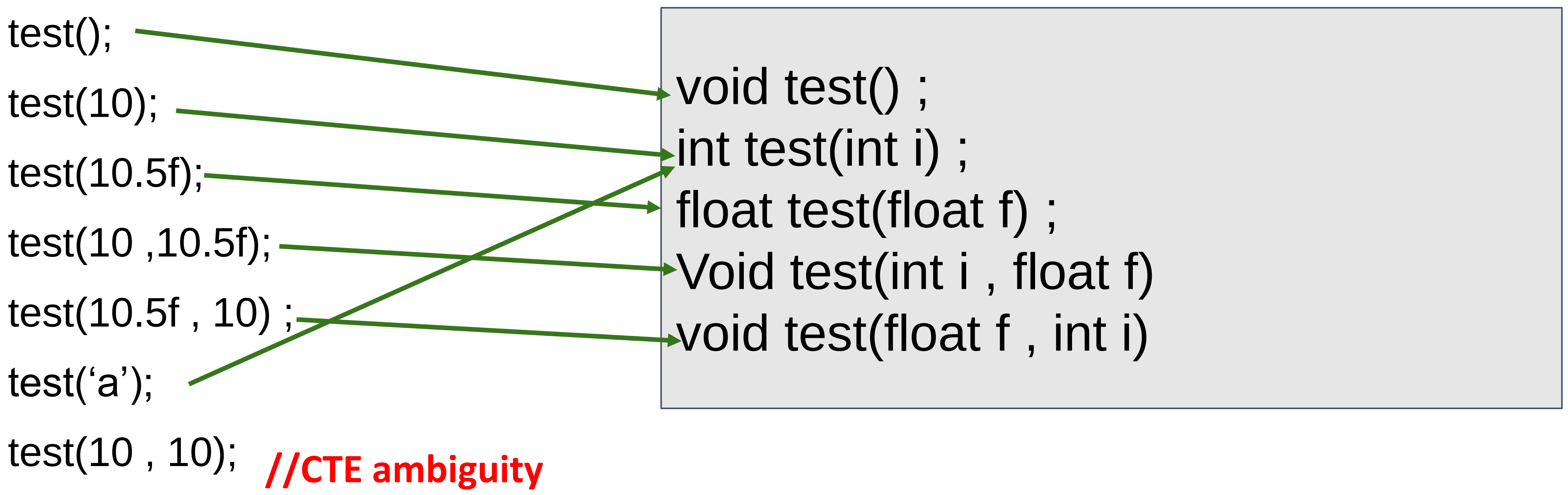
If more than one method is created with the same name but different formal arguments in the same class are known as method overloading.

EXAMPLE : java.lang.Math;

$$\left.\begin{array}{l}\textbf{abs(double d)}\\ \textbf{abs(float f)}\\ \textbf{abs(int i)}\\ \textbf{abs(long l)}\end{array}\right\}\ \textbf{Overloaded method}$$

These are some of the overloaded method (Method with same name but different formal arguments) implemented in java.lang.Math class.

EXAMPLE :



```
test();
test(10);
test(10.5f);
test(10 ,10.5f);
test(10.5f , 10) ;
test('a');
test(10 , 10);   //CTE ambiguity
```

```
void test() ;
int test(int i) ;
float test(float f) ;
Void test(int i , float f)
void test(float f , int i)
```

CONSTRUCTOR OVERLOADING

CONSTRUCTOR OVERLOADING :

A class having more than one constructor with different formal arguments is known as constructor overloading.

METHOD SHADOWING

METHOD SHADOWING :

If a subclass and superclass have the static method with same signature , it is known as method shadowing.

Which method implementation gets execute, depend on what ?

In method shadowing binding is done at compile time , hence it is compile time polymorphism. The execution of the method depends on the reference type and does not depend on the type of object created.

NOTE :

- **Return type should be same**
- **Access modifier should be same or higher visibility than super class method.**
- **Method shadowing is applicable only for the static method.**
- **It is compile time polymorphism**
- **Execution of implemented method depends on the reference type of an object.**

VARIABLE SHADOWING

VARIABLE SHADOWING :

If the superclass and subclass have variables with same name then it is known as variable shadowing.

Which variable is used, depend on what ?

In variable shadowing binding is done at compile time , hence it is a compile time polymorphism. Variable used depends on the reference type and does not depend on the type of object created.

NOTE :

- **It is applicable for both static and non static variable.**
- **It is a compile time polymorphism.**
- **Variable usage depends on type of reference and does not depend on type of object created.**

RUNTIME POLYMORPHISM

RUNTIME POLYMORPHISM :

- If the binding is achieved at the runtime then it is known as runtime polymorphism.
- It is also known as dynamic binding.
- It is achieved by method overriding.

METHOD OVERRIDING

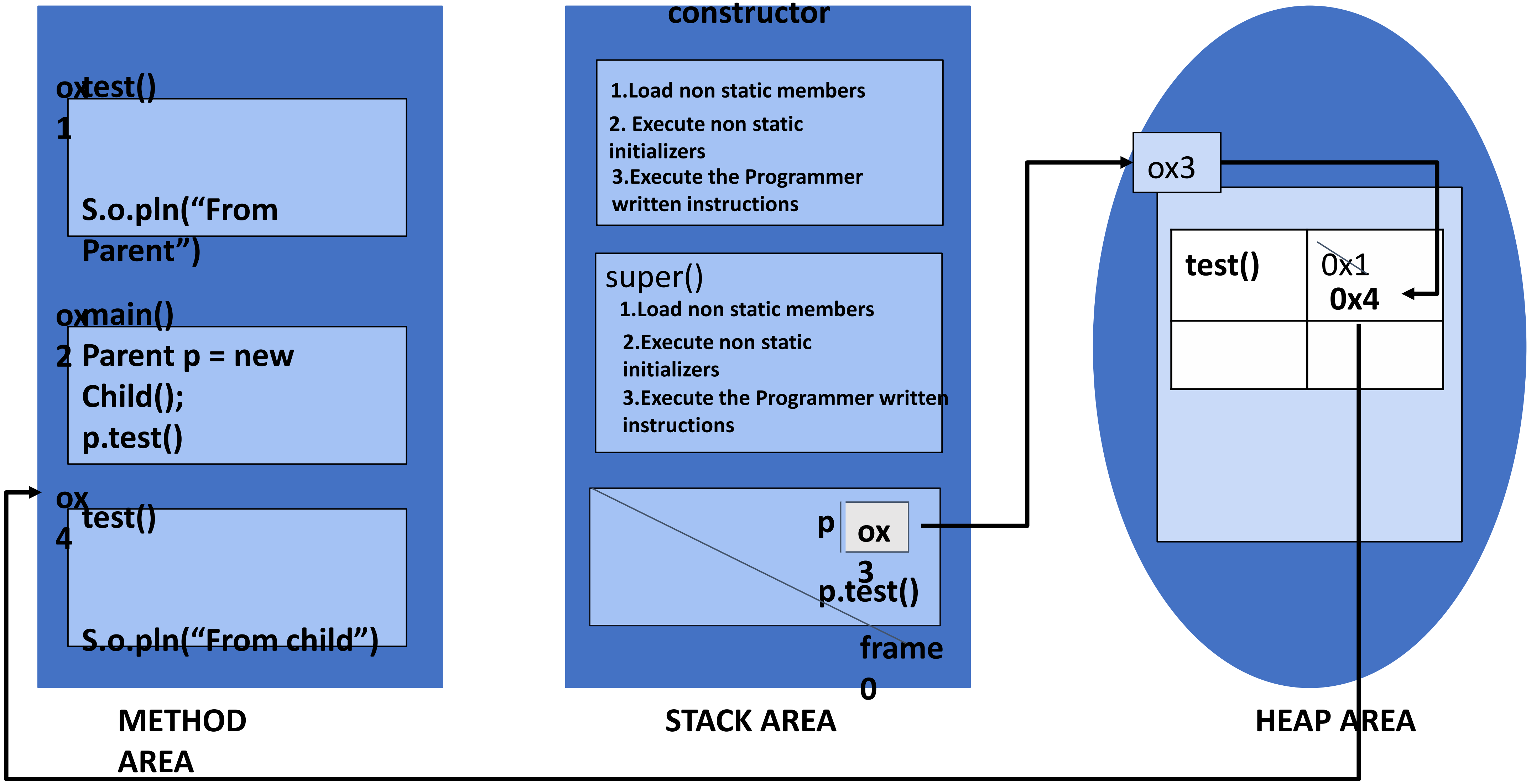
METHOD OVERRIDING :

- If the subclass and superclass have the non static methods with and same signature , it is known as method overriding.

Rule to achieve method overriding :

- **Is-A relationship is mandatory.**
- **It is applicable only for non static methods.**
- **The signature of the subclass method and superclass method should be same.**
- **The return type of the subclass and superclass method should be same until 1.4 version but, from 1.5 version covariant return type in overriding method is acceptable (subclass return type should be same or child to the parent class return type.).**

Method [illegible] static, new register is not created [illegible] instead reference is replaced [illegible] constructor [illegible] called by the Child class [illegible] constructor



ox test()
1
S.o.pln("From Parent")
ox main()
2 Parent p = new Child();
p.test()
ox test()
4
S.o.pln("From child")
1.Load non static members
2. Execute non static initializers
3.Execute the Programmer written instructions
super()
1.Load non static members
2.Execute non static initializers
3.Execute the Programmer written instructions
p ox 3
p.test()
frame 0
ox3
test()
~~0x1~~
0x4
METHOD AREA
STACK AREA
HEAP AREA

EXAMPLE :

```
class Parent
{
        public void test()
        {
                        System.out.println("From parent");
        }
}
class Child extends Parent
{
        @override
        public void test()
        {
                        System.out.println("From child");
        }
        public static void main(String[] args)
        {
                        Parent p = new Child();
                        p.test();
        }
}
```

Why do we need method overriding ?

Method overriding is used to either modify or provide new design for an already existing inherited

EXAMPLE :

```
class AppV1
`       {
                public void feature1()
                {
                        System.out.println(Oldest);
                }
        }
        class AppV2 extends AppV1
        {
                public void feature2()
                {
                        System.out.println(“New feature added”);
                }
                public void feature1()
                {
                        System.out.println(“Updated feature”);
                }
        }
Appv1 app = new Appv2();
```

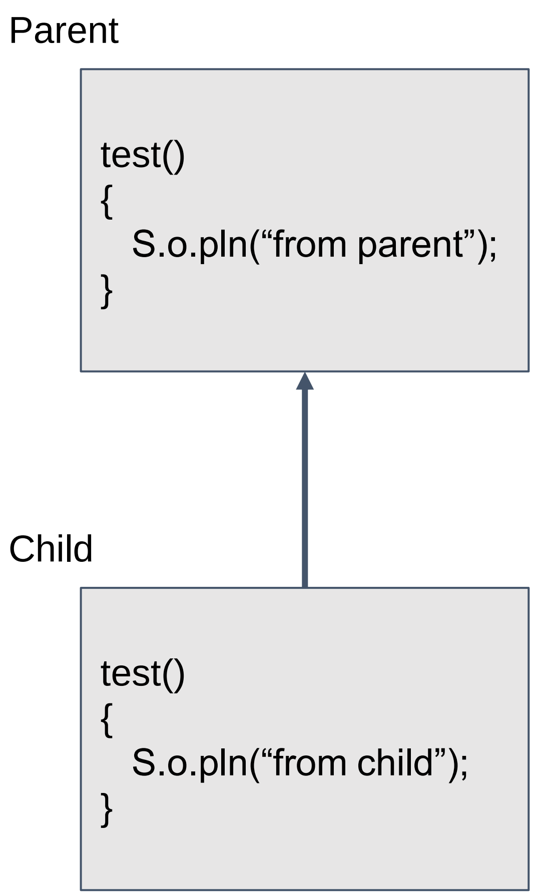
EXAMPLE :

```
Child c = new Child();
c.test(); // from child
Parent p = c;
p.test(); // from child
```

Internal runtime object is child so child test() will get executed, it does not depend on the reference type.

NOTE :

Variable overriding is not applicable.



FINAL MODIFIER

FINAL MODIFIER :

- final modifier indicates that an object is fixed and can’t be changed.
- It is applicable to variables , method , and class level object.

FINAL VARIABLE

FINAL VARIABLE :

If a variable is prefixed with final modifier then the value of the variable remains constant once it gets initialized.

NOTE :

If we try to change / modify the value of final variable then we will get compile time error.

EXAMPLE :

```
final int i = 10 ;
i = 20 ; //CTE
final int j ;
j = 30 ; // initializatIon
j = 40 ; //CTE
```

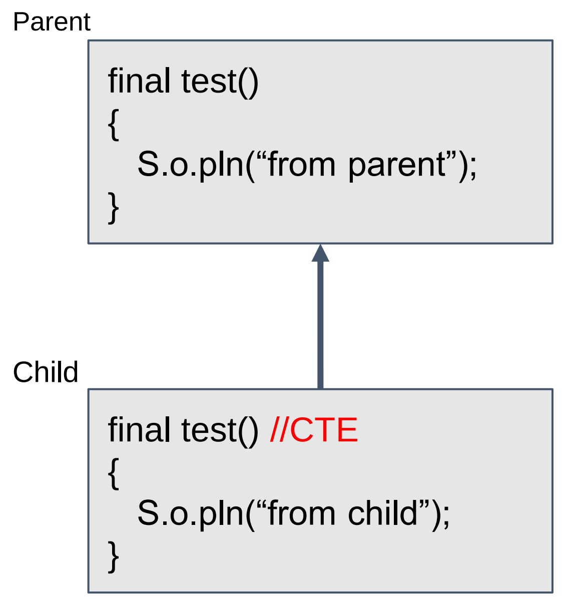
FINAL METHOD

FINAL METHOD:

If a method if prefixed with final modifier then overriding of that method is not possible.

NOTE :

If we try to override that method we will get compile time error.



```
final test()
{
   S.o.pln(“from parent”);
}
```

```
final test() //CTE
{
   S.o.pln(“from child”);
}
```

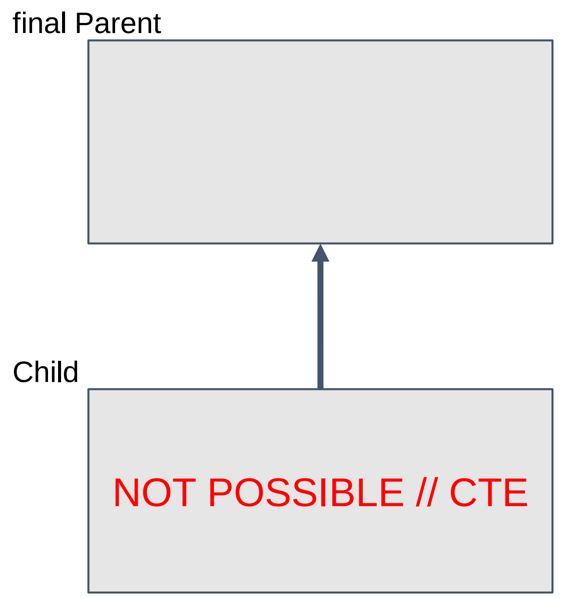
FINAL CLASS

FINAL CLASS :

If the class is prefixed with final modifier ithen we can't inherit from that class is not possible).

NOTE :

If we try to create a child class for the final class then we will get compile time error.



INTRODUCTION TO OBJECT CLASS

Object class :

- Object class is defined in java.lang package.
- Object class is a supermost parent class for all the classes in java.
- In object class there are 11 non static methods.

METHODS IN OBJECT CLASS

1	public String toString()
2	public boolean equals(Object o)
3	public int hashCode()
4	protected Object clone() throws CloneNotSupportedException
5	protected void finalize()
6	final public void wait() throws InterruptedException
7	final public void wait(long l) throws InterruptedException
8	final public void wait(long l , int i) throws InterruptedException
9	final public void notify() throws InterruptedException
10	final public void notifyAll() throws InterruptedException
11	final public void getClass()

toString() METHOD

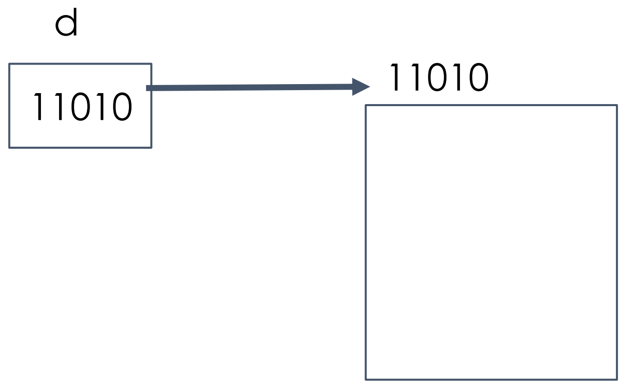
toString()

- toString() method returns String.
- toString() implementation of Object class returns the reference of an object in the String format.

Return Format : ClassName@HexaDecimal

EXAMPLE :

```
class Demo
{
        public static void main(String[] args)
        {
                Demo d = new Demo();
                System.out.println(d) // d.toString() --- Demo@2f92e0f4
        }
}
```



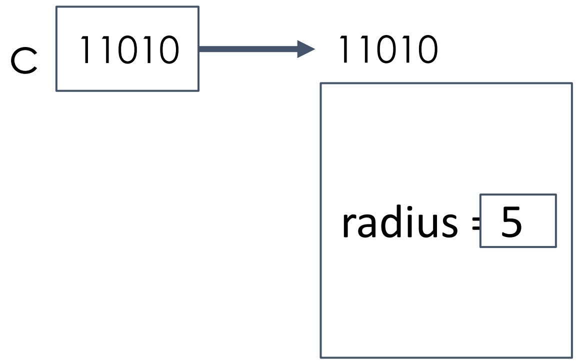
OVERRIDING toString()

PURPOSE OF OVERRIDING toString() :

We override toString() method to print state of an object instead of printing reference of an object.

EXAMPLE :

```
class Circle
{
        Circle(int radius)
        {
                this.radius = radius ;
        }
        Int radius ;
        @Override
        Public String toString()
{
        return "radius : "+radius ;
}
Public static void main(String[] args)
{
        Circle c = new Circle(5);
        System.out.println(c) // c.toString()---radius : 5
}
}
```



NOTE :

- Java doesn’t provide the real address of an object.
- Whenever programmer tries to print the reference variable toString() is implicitly called.

equals(Object) METHOD

equals(Object) :

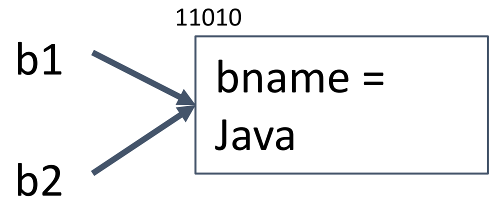
- The return type of equals(Object) method is boolean.
- To equals(Object) method we can pass reference of any object.
- The java.lang.Object class implementation of equals(Object) method is used to compare the reference of two objects.

EXAMPLE 1 :

```
class Book
{
        String bname;
        Book(String bname)
        {
                this.bname = bname;
        }
}
```

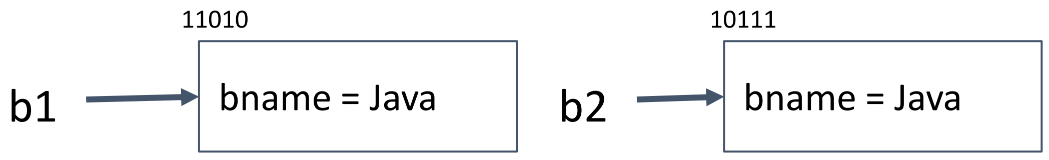
Case 1

```
Book b1 = new Book("Java");
Book b2 = b1;
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // true
s.o.pln(b1.equals(b2)); // true
```



Case 2

```
Book b1 = new Book("Java");
Book b2 = new Book("Java");
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // false
s.o.pln(b1.equals(b2)); // flase
```



NOTE :

- If the reference is same == operator will return true else it returns false.
- The equals(Object) method is similar to == operator.

OVERRIDING equals(Object)

PURPOSE OF OVERRIDING equals(Object) :

We override to equals(Object) method to compare the state of an two Objects instead of comparing reference of two Objects.

NOTE :

- **If equals(Object) method is not overridden it compares the reference of two objects similar to == operator.**
- **If equals(Object) method is overridden it compares the state of two objects, in such case comparing the reference of two objects is possible only by == operator.**

Design tip :

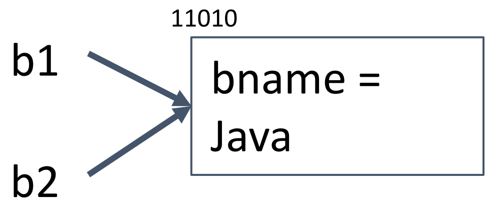
In equals method compare the state of an current(this) object with the passed object by downcasting.

EXAMPLE :

```
class Book
{
        String bname ;
        Book(String bname)
        {
                this.bname = bname ;
        }
        @Override
        public boolean equals(Object o)
        {
                Book b = (Book)o;
                return  this.bname.equals( b.bname) ;
        }
}
```

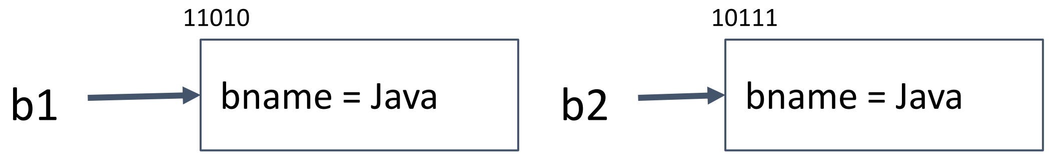
Case 1

```
Book b1 = new Book("Java");
Book b2 = b1;
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // true
s.o.pln(b1.equals(b2)); // true
```



Case 2

```
Book b1 = new Book("Java");
Book b2 = new Book("Java");
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // false
s.o.pln(b1.equals(b2)); // true
```



hashCode() METHOD

hashCode() :

- The return type of hashCode() method is int.
- The java.lang.Object implementation of hashCode() method is used to give the unique integer number for every object created.
- The unique number generated based on the reference of an object.

OVERRIDING hashCode()

PURPOSE OF OVERRIDING hashCode() :

If the equals(Object) method is overridden , then it is necessary to override the hashCode() method.

Design tip :

hashCode() method should return an integer number based on the state of an object.

EXAMPLE 1:

```
class Pen
{
        double price ;
        Pen(double price)
        {
                this.price = price ;
        }
        @Override
        public int hashCode()
        {
                int hc = (int)price;
                return hc ;
        }
}
```

EXAMPLE 2 :

```
class Book
{
    int bid ;
    double price ;
    Book(int bid , double price)
    {
            this.bid = bid ;
            this.price = price;
    }
    @Override
    public int hashCode()
    {
            int hc1 = bid ;
            double hc2 = price ;
            int hc = hc1+(int)hc2;
            return hc ;
    }
}
```

EXAMPLE 1 :

```
class Book
{
        String bname;
        Book(String bname)
        {
                this.bname = bname;
        }
}
```

Case 1

```
Book b1 = new Book("Java");
Book b2 = b1;
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // true
S.o.pln(b1.equals(b2)); // true
int h1 = b1.hashCode();
Int h2 = b2.hashCode();
S.o.pln(h1==h2); // true
```

Case 2

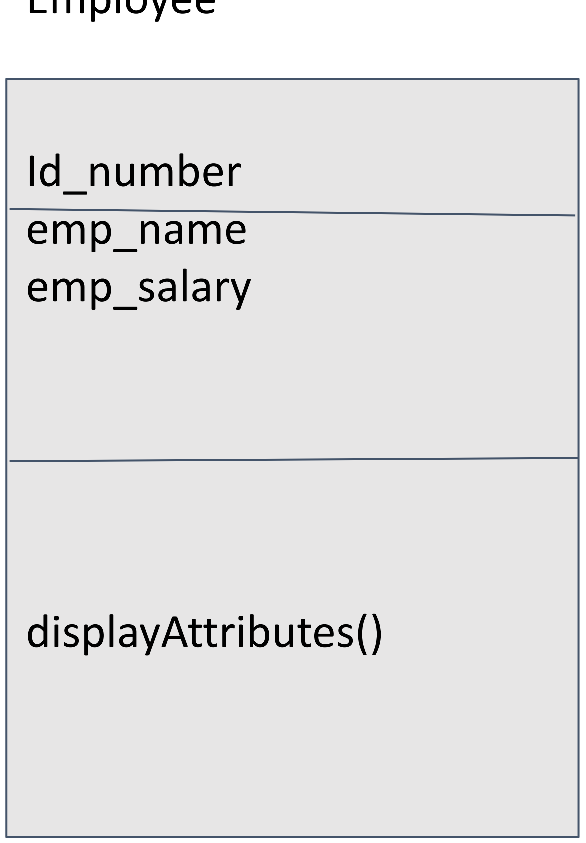
```
Book b1 = new Book("Java");
Book b2 = new Book("Java");
S.o.pln(b1.bname); // Java
S.o.pln(b2.bname); // Java
S.o.pln(b1==b2); // false
S.o.pln(b1.equals(b2)); // flase
int h1 = b1.hashCode();
Int h2 = b2.hashCode();
S.o.pln(h1==h2); // false
```

OBSERVATION :

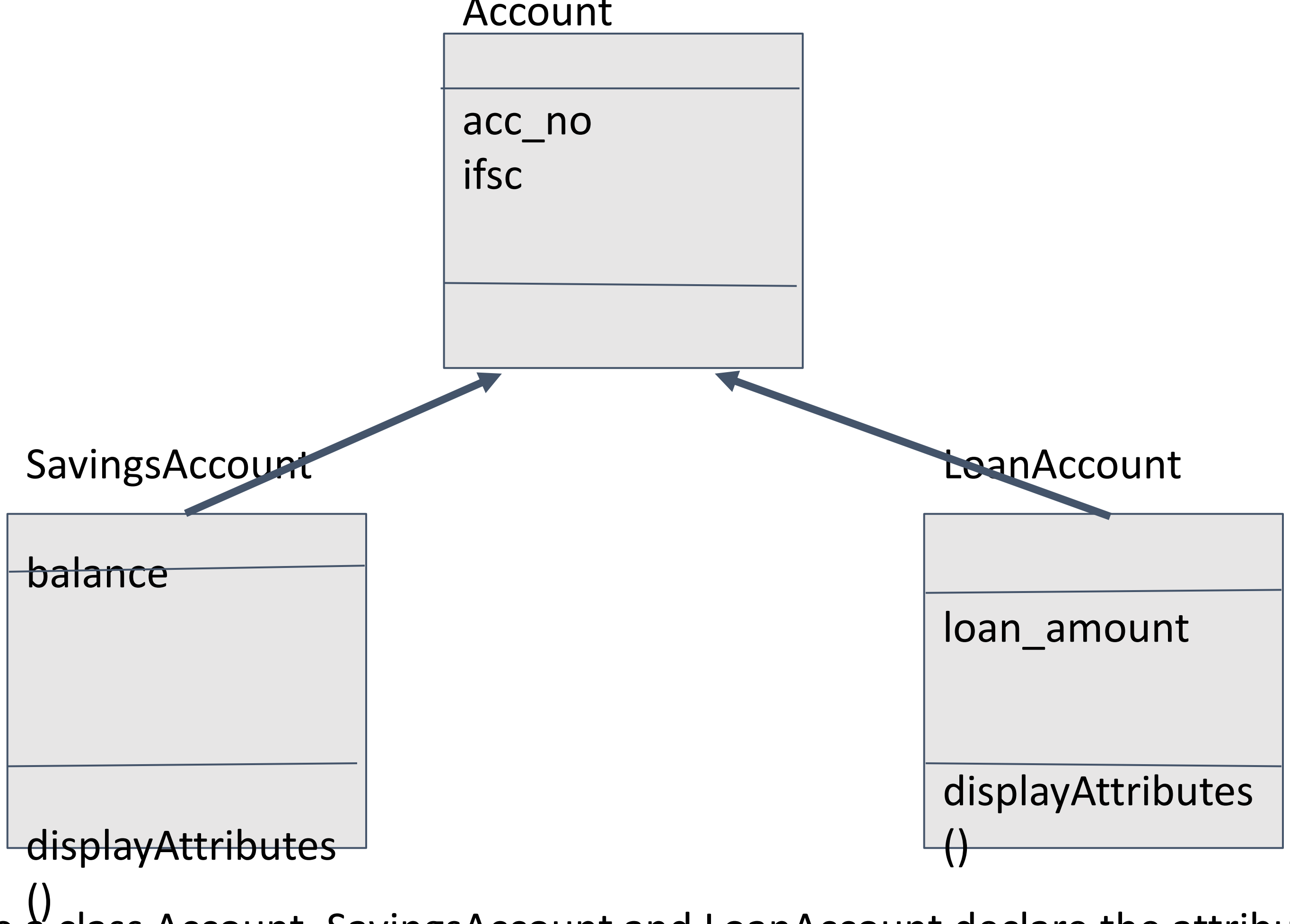
In the above two cases it is clear that,

- If the hashcode for two object is same, equals(Object) method will return true.
- If the hashcode for two object is different, equals(Object) method will return false.

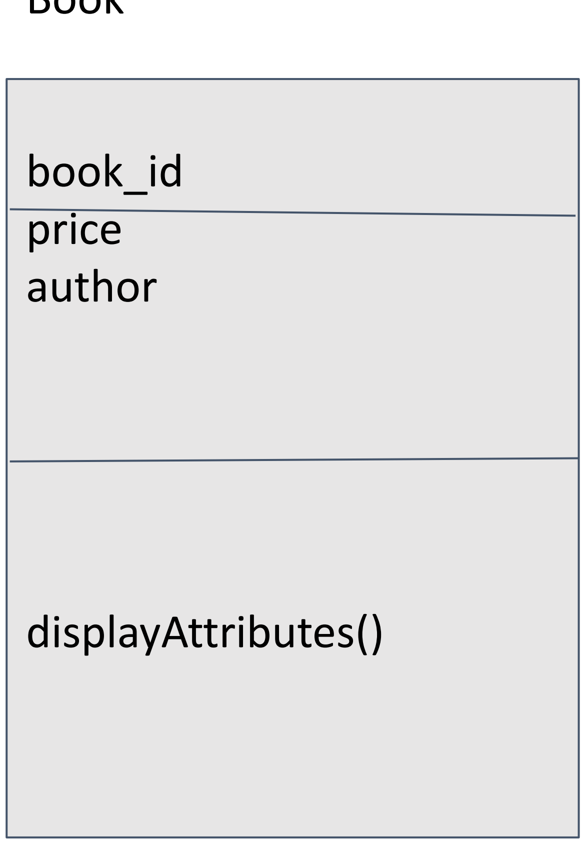
PROGRAMMING SESSION ON OBJECT CLASS



Create a class Employee , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Employee class.
Override the toString(), equals() and hashCode().

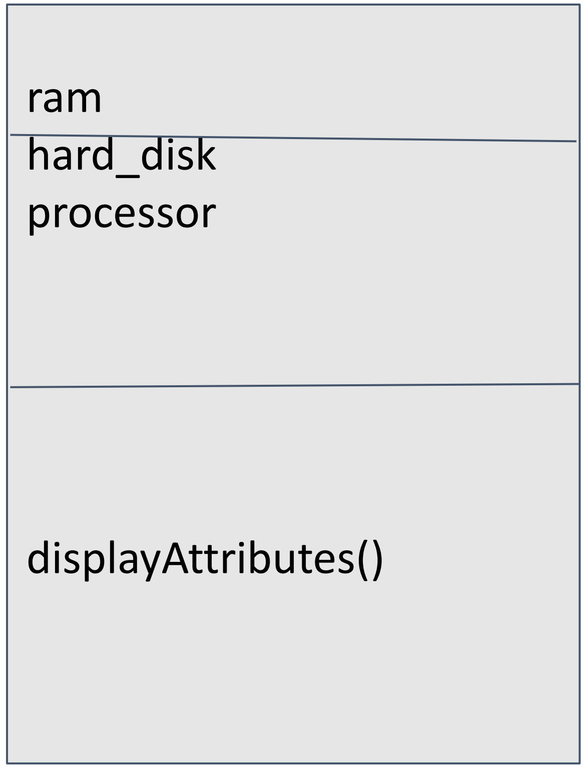


Create a class Account, SavingsAccount and LoanAccount declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for all the class.
Override the toString(), equals() and hashCode().



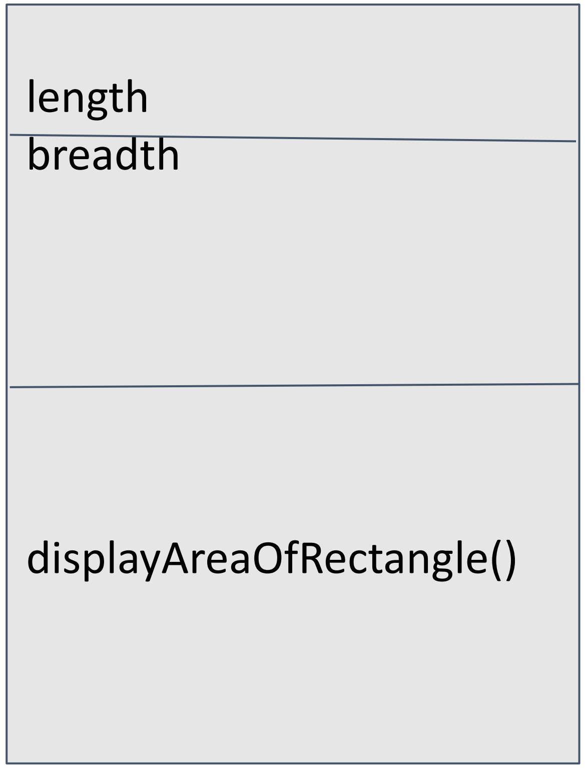
Create a class Book , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Book class.
Override the toString(), equals() and hashCode().

Laptop



Create a class Laptop , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Laptop class.
Override the toString(), equals() and hashCode().

Rectangle



Create a class Rectangle , declare the attributes as private.
Create a getter and setter methods for attributes.
Create a necessary constructors for the Rectangle class.
Override the toString(), equals() and hashCode().

STRING

STRING :

String is a literal (data). It is a group of character that is enclosed within the double quote " ".

- It is a Non primitive data.
- In java we can store a string by creating instance of the following classes.
    - **java.lang.String**
    - **java.lang.StringBuffer**
    - **java.lang.StringBuilder**
- In java, whenever we create a string compiler implicitly create an instance for java.lang.string in string pool area / string constant pool (scp).

STRING LITERAL

STRING LITERAL:

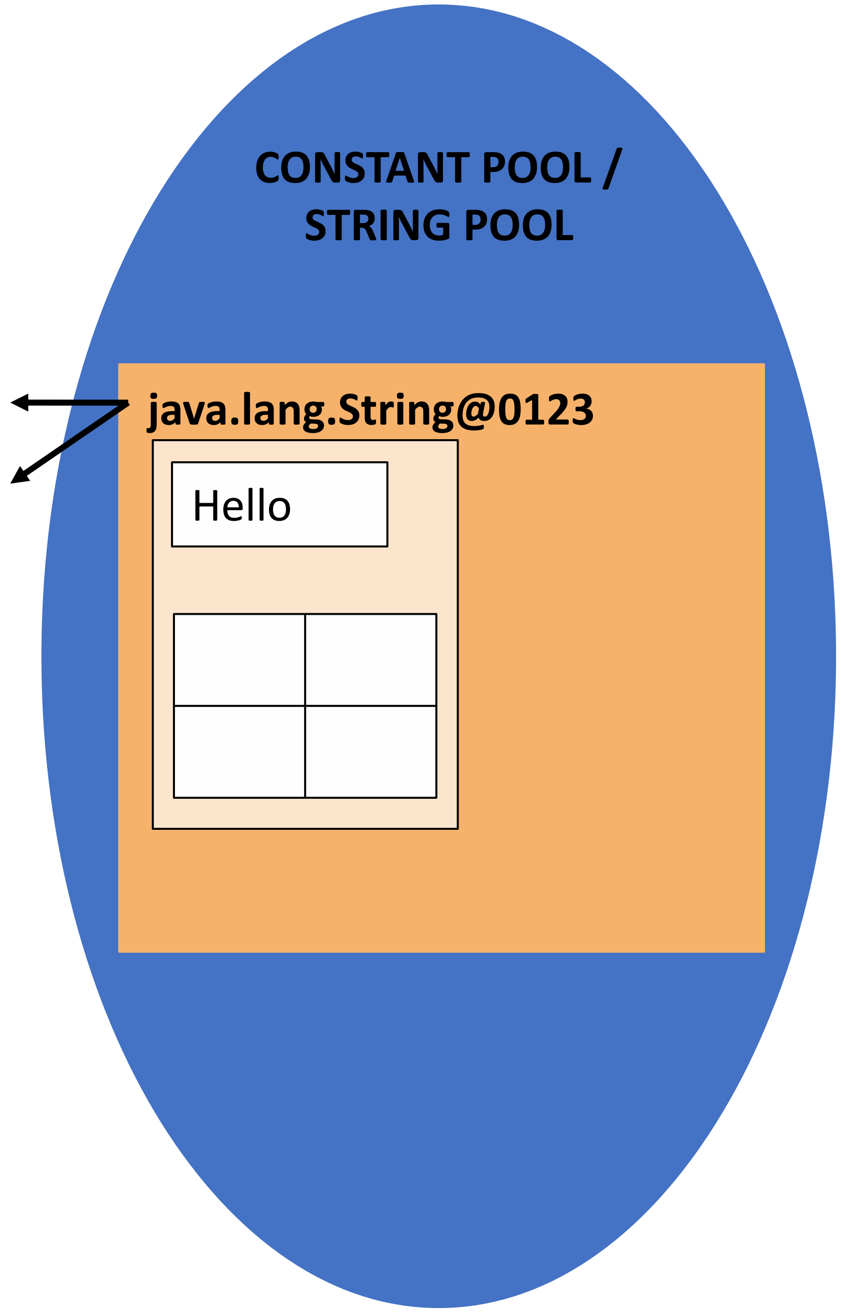
Anything enclosed within the double quote “ ” in java is considered as String literal.

Characteristics of String Literal :

- When a String literals is used in a java program, an instance of java.lang.String class is created inside a String pool.
- For the given String literal, If the instance of a String is already present, then new instance is not created instead the reference of a existing instance is given.

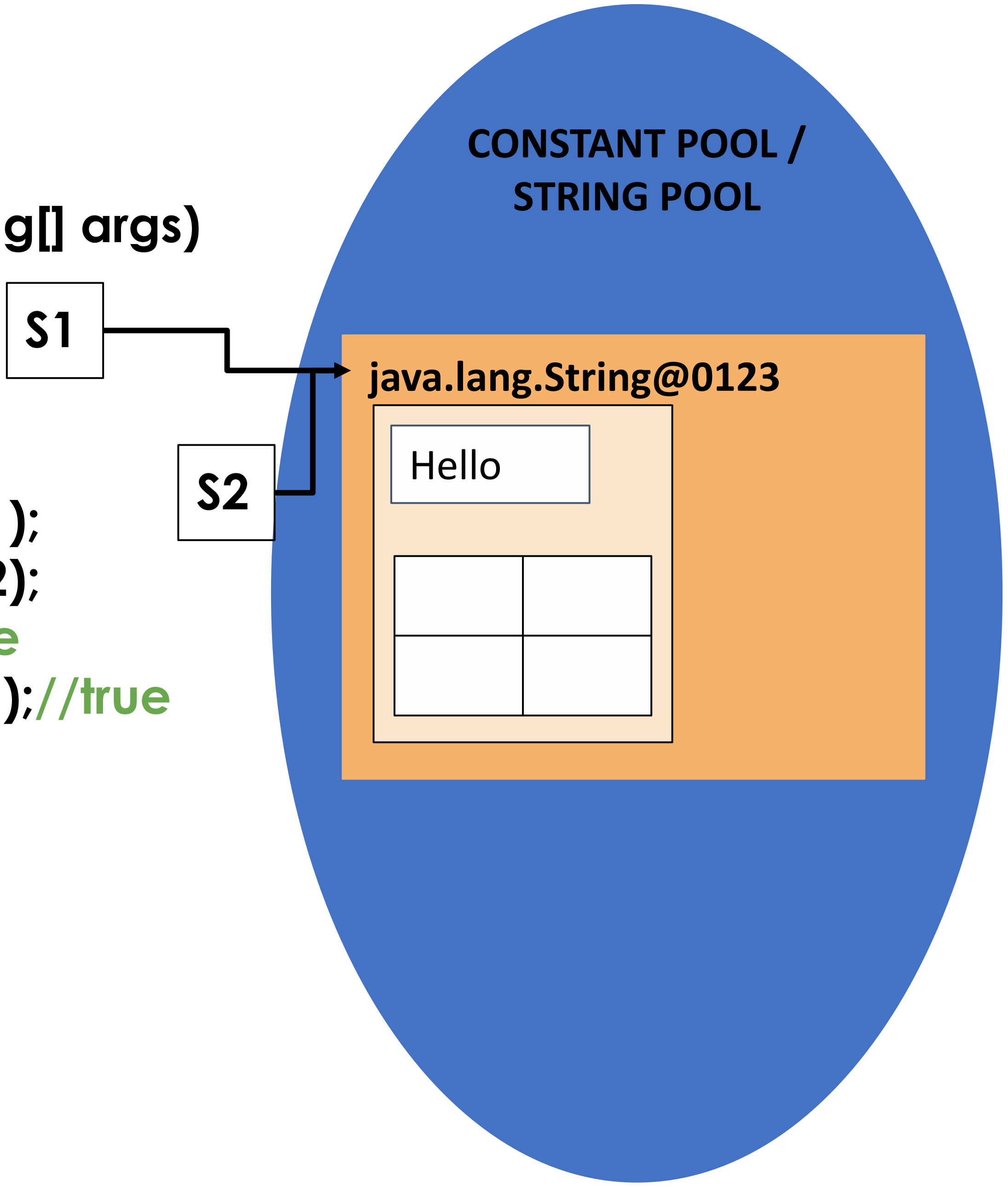
EXAMPLE:1

```
Class Demo
{
        public static void main(String[] args)
        {
                System.out.println("Hello"); //String@0123
                System.out.println("Hello"); //String@0123
        }
}
```



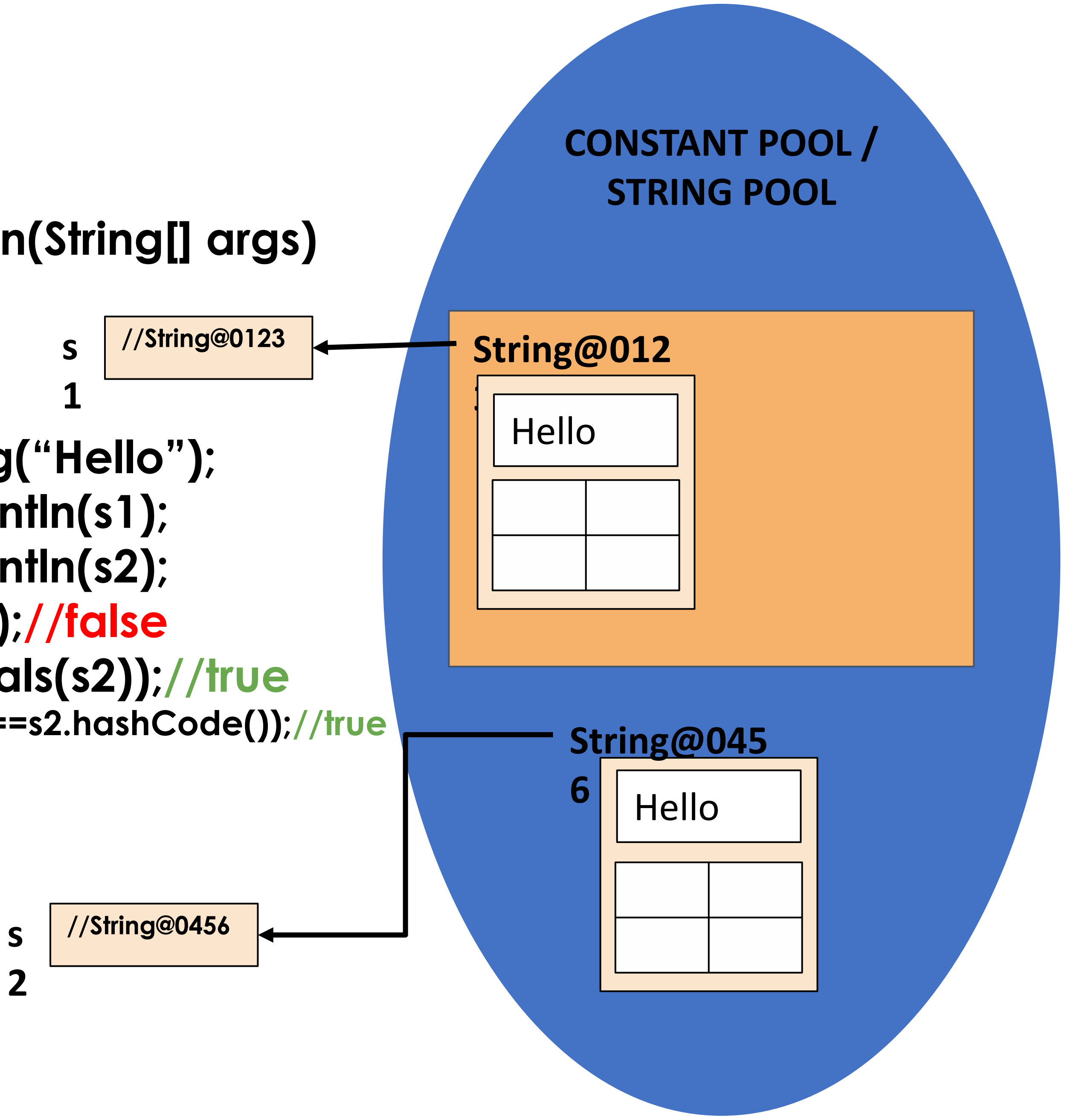
EXAMPLE:2

```
Class Demo
{
        public static void main(String[] args)
        {
                String s1, s2;
                s1 = “Hello”;
                s2 = “Hello”;
                System.out.println(s1);
                System.out.println(s2);
    System.out.println(s1==s2);//true
    System.out.println(s1.equals(s2));//true
}
}
```



EXAMPLE:3

```
Class Demo
{
        public static void main(String[] args)
        {
                String s1, s2;
                s1 = "Hello";
                s2 = new String("Hello");
                System.out.println(s1);
                System.out.println(s2);
    System.out.println(s1==s2);//false
    System.out.println(s1.equals(s2));//true
    System.out.println(s1.hashCode()==s2.hashCode());//true
}
}
```



STRING CLASS

java.lang.String :

- String is a inbuilt class defined in a java.lang package.
- It is a final class.
- It inherits java.lang.Object
- In a String class toString(), equals(), hashCode() methods of java.lang.Object class are overridden.

It implements :

- Comparable
- Serializable
- CharSequence

CONSTRUCTOR IN STRING CLASS

CONSTRUCTORS :

CONSTRUCTORS	DESCRIPTION
String()	Creates an empty string object
String(String literals)	Creates string object by initializing with string literals
String(char[] ch)	Creates String by converting character array into string
String(byte[] b)	Creates String by converting byte array into string
String(StringBuffer sb)	Creates String by converting StringBuffer to string
String(StringBuilder sb)	Creates String by converting StringBuilder to string

METHODS OF STRING CLASS

IMPORTANT METHODS :

RETURN TYPE	METHOD NAME	DESCRIPTION
String	toUpperCase()	Converts the specified string to Upper case
String	toLowerCase()	Converts the specified string to Lowercase
String	concat(String s)	joins the specified Strings
String	trim()	Remove the space present before and after the string
String	substring(int index)	Extract a characters from a string object starts from specified index and ends at the end of a string
String	substring(int start, int end)	Extract a characters from a string starts from specified index and ends at end-1 index

RETURN TYPE	METHOD NAME	DESCRIPTION
char	**charAt(int index)**	Returns character of the specified index from the string
int	**indexOf(char ch)**	Return the index of the character specified if not return -1
int	**indexOf(char ch, int Start_Index)**	Return the index of the character specified by searching from specified index if not return -1
int	**indexOf(charSequence str)**	Return the index of the specified string index if not return -1
int	**indexOf(charSequence str,int Start_Index)**	Return the index of the specified string by searching from specified index if not returns -1
int	**lastIndexOf(char ch)**	Returns the index of the character which is occurred at last in the original String
int	**length()**	Returns length of the string

RETURN TYPE	METHOD NAME	DESCRIPTION
boolean	**equals(Object o)**	Compares states of a two strings
boolean	**equalsIgnoreCase(String s)**	Compares two strings by ignoring its case
boolean	**contains(String str)**	Returns true if specified String is present else it returns false
boolean	**isEmpty()**	Returns true if string is empty else return false
char[]	**toCharArray(String str)**	Converts the specified string into character array
string[]	**split(String str)**	Break the specified string into multiple string and returns String array
byte[]	**getBytes()**	Converts the specified string to byte value and returns byte array

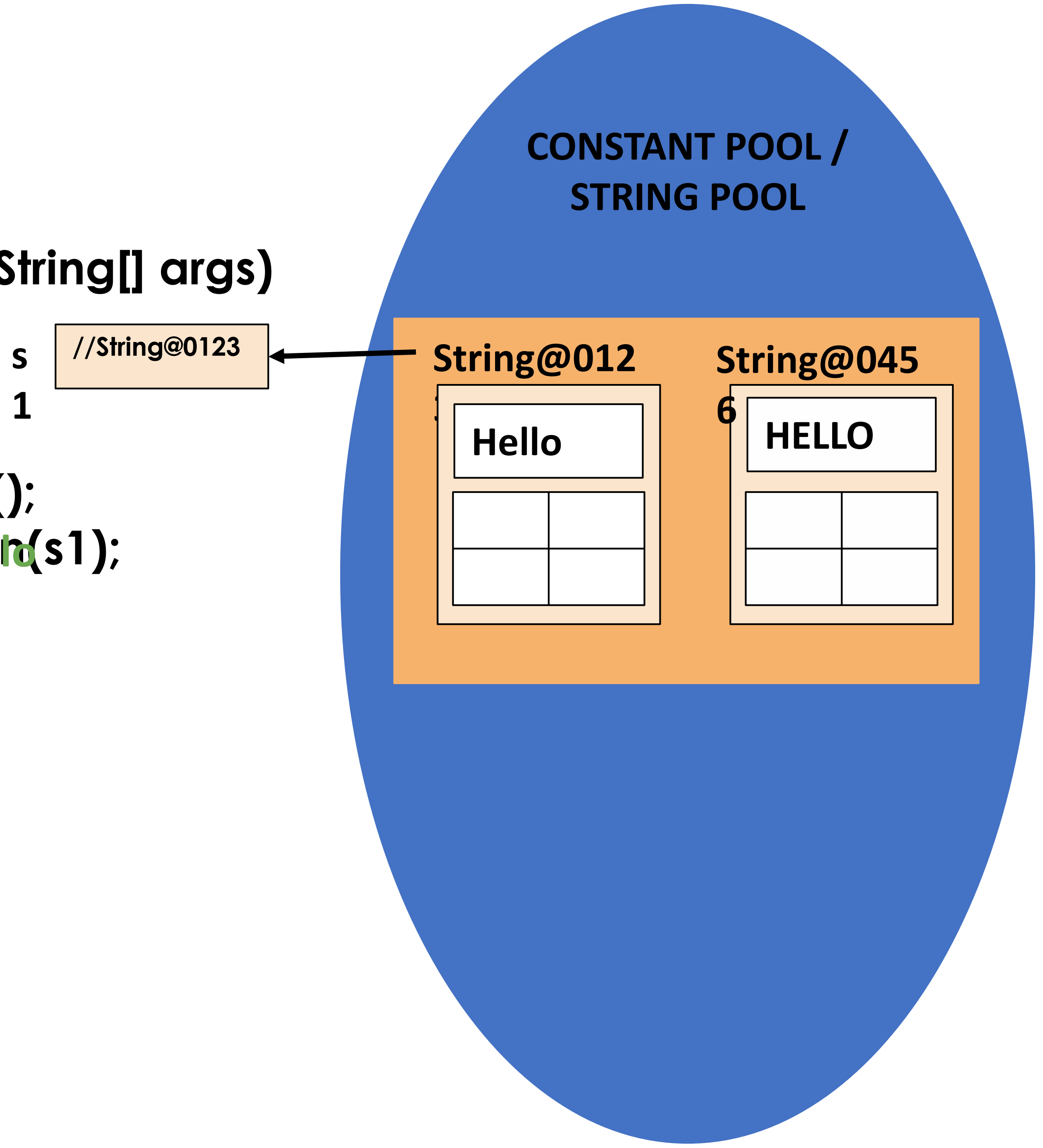
CHARACTERISTICS OF STRING

CHARACTERISTIC :

- Instance of String class is immutable in nature. (Once the object is created then the state is not modified)
- If we try to manipulate (Modify) the state/data then new object is created and reference is given

EXAMPLE:1

```
Class Demo
{
        public static void main(String[] args)
        {
                String s1, s2;
                s1 = "Hello";
                s1.toUpperCase();
                System.out.println(s1); // Hello
}
}
```



COMPARISON OF STRING

COMPARISON OF STRING :

- **== ------------------------->**Compares the reference
- **equals() ----------------->**Compares state/data of the object
- **equalsIgnoreCase()--->**Compares the state/data of the object by ignoring its case
- **compareTo()------------>**Compares two string and returns integer value

  Syntax: "String1".compareTo("String2")

  - **string1==string2 ---> 0**
  - **string1>string2-------> +ve Integer**
  - **string1<string2--------> -ve Integer**

DISADVANTAGE OF java.lang.String

DISADVANTAGES :

Immutability, because for every modification separate object is get created in a memory, it reduces the performance.

NOTE :

To overcome the disadvantage of String class we can go for StringBuffer and StringBuilder

EXAMPLE:2 Reversing of a String

```
Class Demo
{
        public static void main(String[] args)
        {
                String s1=”Cat”;
                String reverse = “”;
                for(int i=s1.length()-1; i>=0; i--)
                {
                        reverse = reverse+s1.charAt(i);
                }
                System.out.println(reverse);
}
}
```

Iteration 2:

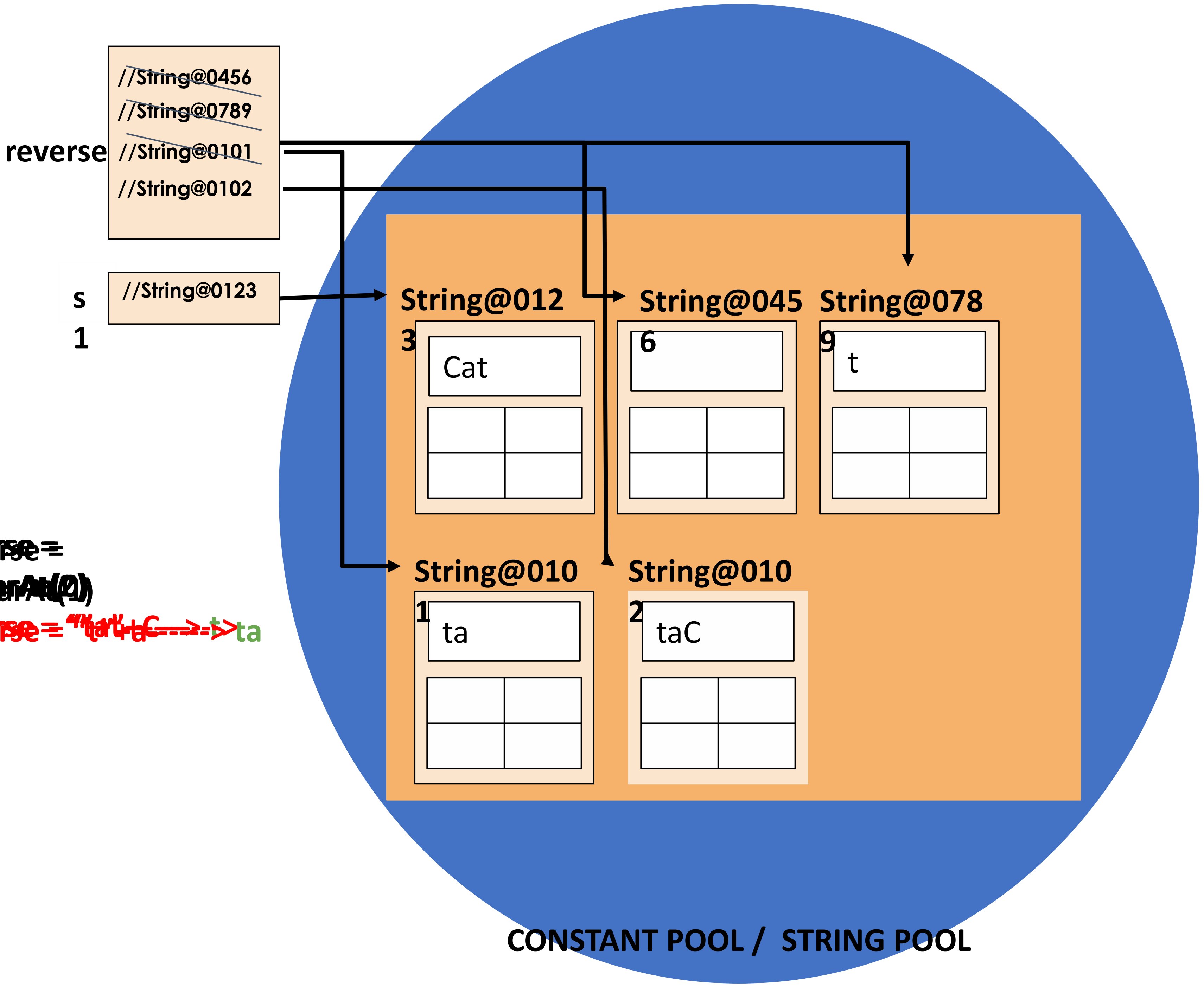
Iteration 3:

reverse = reverse+S1.charAt(1)

reverse = reverse+S1.charAt(2)

reverse = "t"+a ---> ta

reverse = "ta"+C ---> taC



STRINGBUFFER CLASS

java.lang.StringBuffer :

- It is a inbuilt class defined in java.lang package.
- It is a final class.
- It helps to create mutable instance of String.
- StringBuffer does not have SCP.
- It inherits java.lang.Object class.
- In StringBuffer equals(), hashcode() methods of java.lang.Object class are not overridden.

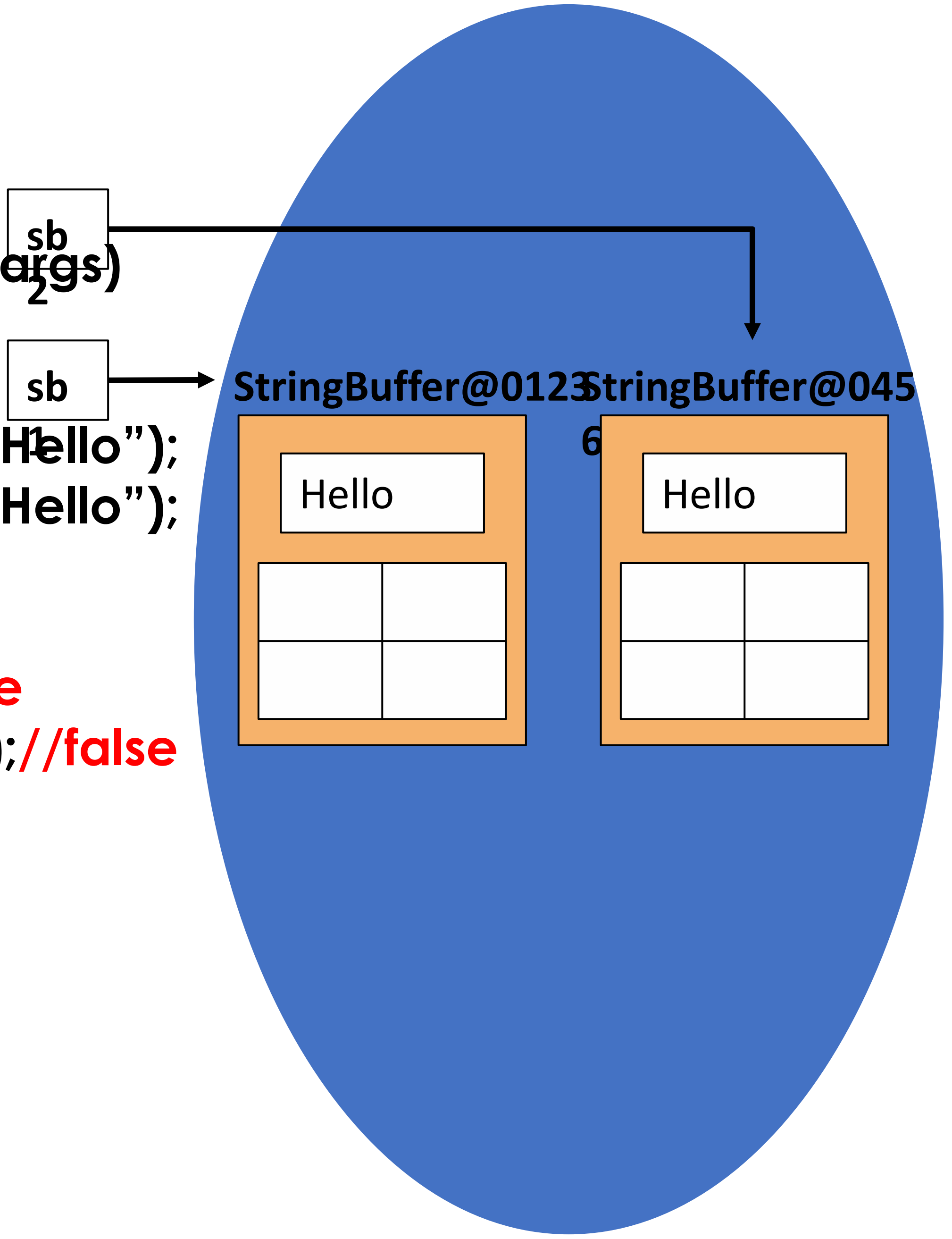
It implements :

Serializable

CharSequence

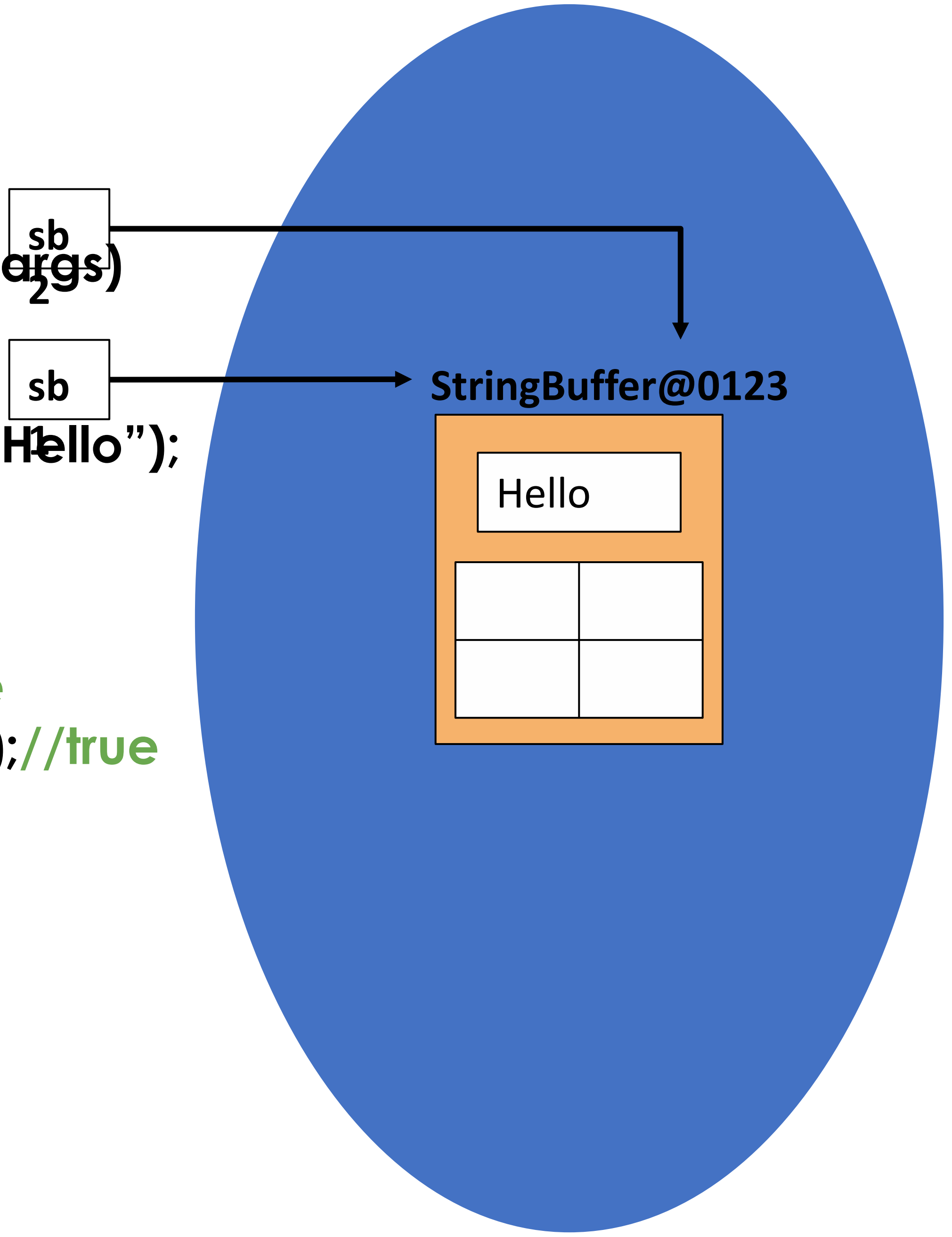
EXAMPLE : 1

```
Class Demo
{
        public static void main(String[] args)
        {
                StringBuffer sb1, sb2;
                sb1 = new StringBuffer("Hello");
                sb2 = new StringBuffer("Hello");
                System.out.println(sb1);
                System.out.println(sb2);
     System.out.println(sb1==sb2);//false
     System.out.println(sb1.equals(sb2));//false
}
}
```



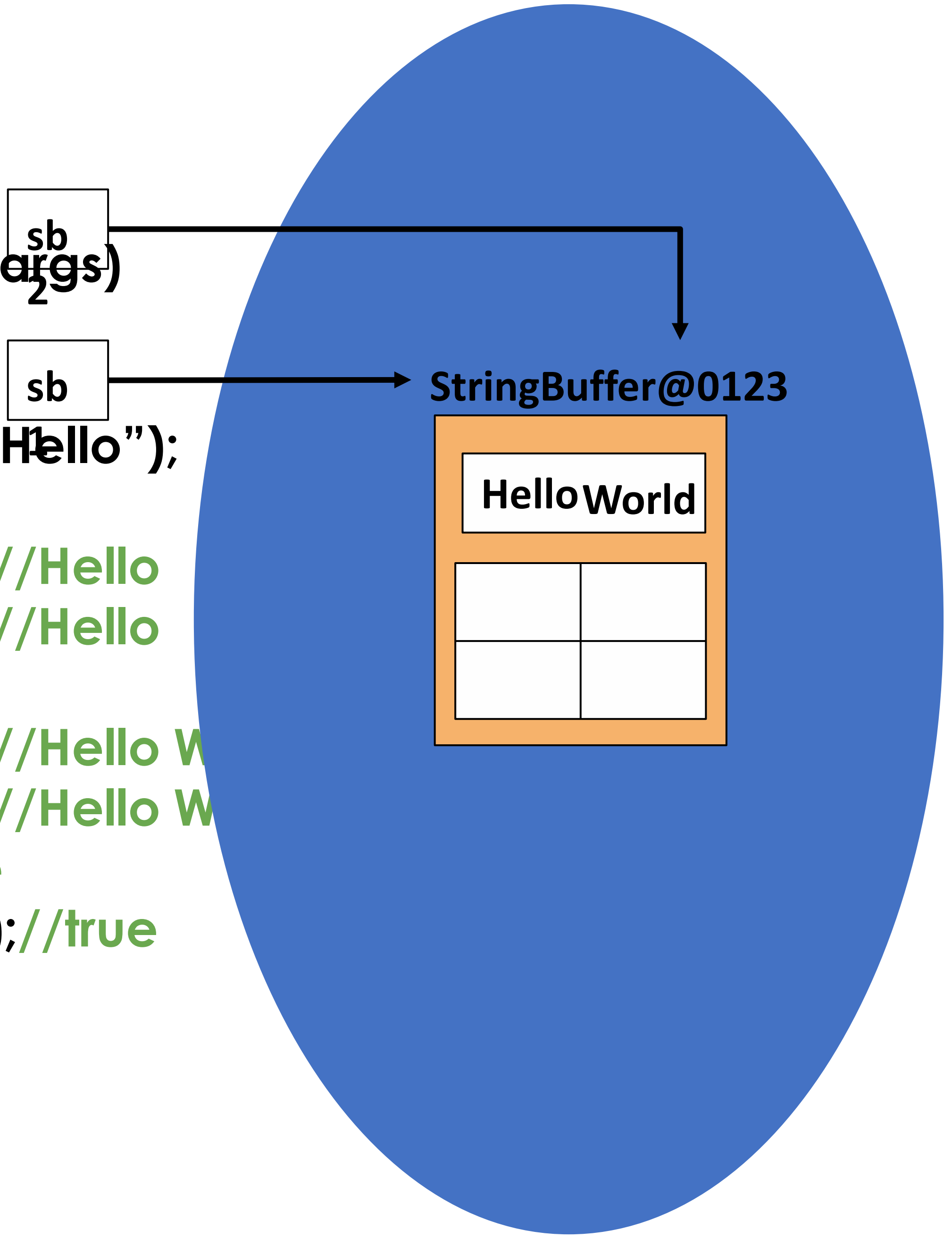
EXAMPLE : 2

```
Class Demo
{
        public static void main(String[] args)
        {
                StringBuffer sb1, sb2;
                sb1 = new StringBuffer("Hello");
                sb2 = sb1;
                System.out.println(sb1);
                System.out.println(sb2);
    System.out.println(sb1==sb2);//true
    System.out.println(sb1.equals(sb2));//true
}
}
```



EXAMPLE : 3

```
Class Demo
{
        public static void main(String[] args)
        {
                StringBuffer sb1, sb2;
                sb1 = new StringBuffer("Hello");
                sb2 = sb1;
                System.out.println(sb1);//Hello
                System.out.println(sb2);//Hello
                sb1.append(" World");
                System.out.println(sb1);//Hello W
                System.out.println(sb2);//Hello W
    System.out.println(sb1==sb2);//true
    System.out.println(sb1.equals(sb2));//true
}
}
```



CONSTRUCTORS

CONSTRUCTORS	DESCRIPTION
StringBuffer()	Creates empty String with initial capacity 16
StringBuffer(String str)	Creates string buffer with the specified string

IMPORTANT METHODS OF STRINGBUFFER

METHODS :

RETURN TYPE	METHOD NAME	DESCRIPTION
int	capacity()	Returns current capacity.
int	length()	Returns length of the string.
char	charAt(int index)	Returns the character of the specified index.
StringBuffer	append(String s)	Join strings. (Overloaded method).
StringBuffer	insert(int index, String s)	Insert a specified string into original String of specified index. (Overloaded method)
StringBuffer	delete(int begin, int end)	Delete String from specified beginning index to end-1 index.

RETURN TYPE	METHOD NAME	DESCRIPTION
StringBuffer	deleteCharAt(int index)	Delete the character present in the specified index.
StringBuffer	reverse()	Reverse the string
StringBuffer	setLength(int length)	Only specified length in a string is exist remaining get removed.
StringBuffer	substring(int begin)	Returns the substring from the specified beginning index
StringBuffer	substring(int begin, int end)	Returns substring from specified beginning index to end-1 index.
StringBuffer	replace(int begin,int end,String s)	Replace a specified string from the beginning index to end-1 index
void	trimToSize()	Removes the unused capacity or set the capacity till length of the string
void	setCharAt(int index, char new char)	Replace the new character in a string of specified index.
void	ensureCapacity(int capacity)	Sets capacity for storing a string.

CHARACTERISTICS OF STRINGBUFFER

CHARACTERISTICS :

It is immutable.

NOTE :

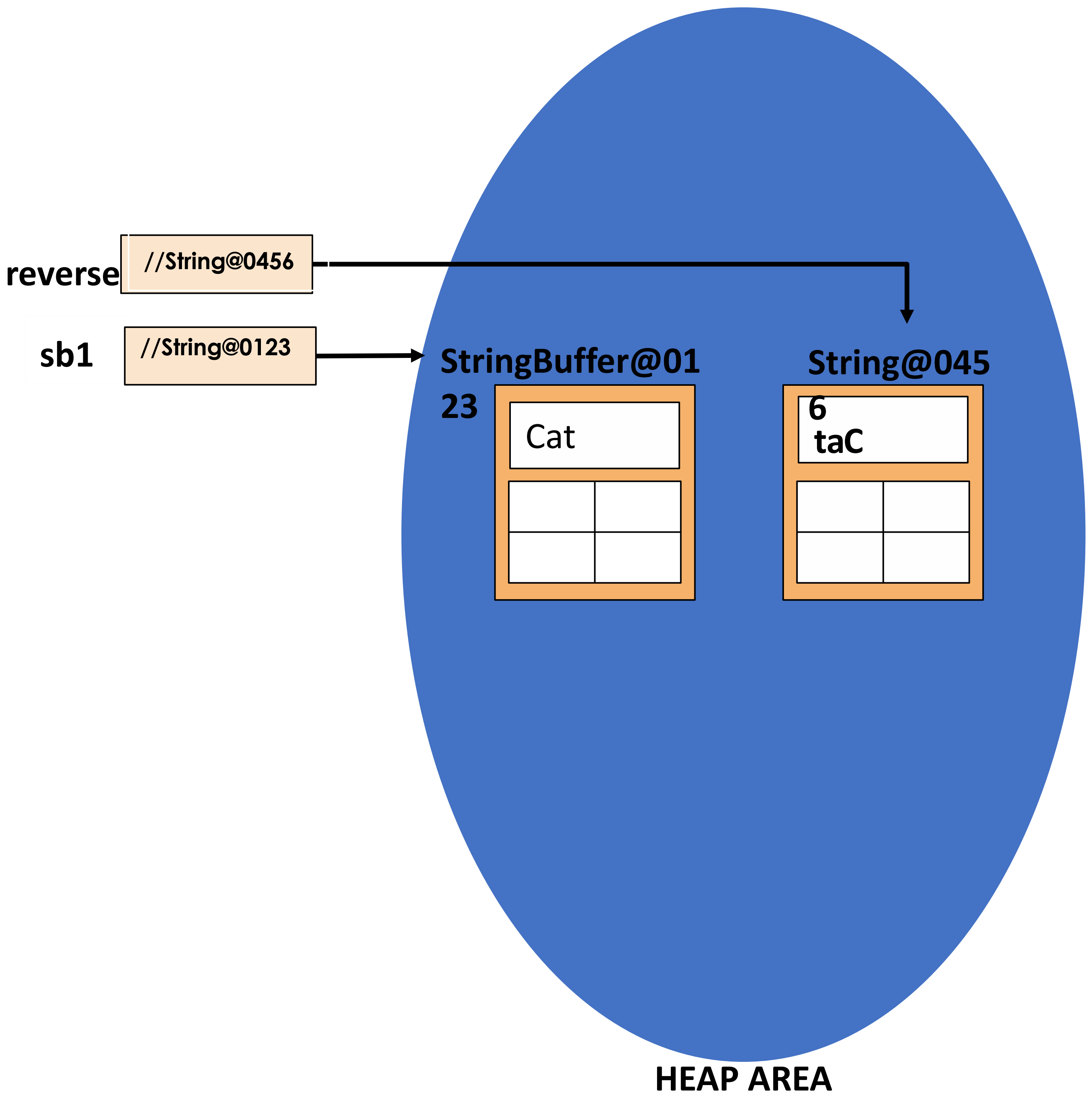
String constant pool is not applicable to String Buffer.

EXAMPLE :

```
class StringBufferReverseDemo
{
        public static void main(String[] args)
        {
                StringBuffer sb1=new StringBuffer("Priyatharsan");
                StringBuffer reverse = new StringBuffer();
                for(int i=sb1.length()-1; i>=0 ; i--)
                {
                        reverse.append(sb1.charAt(i));
                }
                System.out.println(reverse);
        }}
```

Iteration 3:

reverse.append(S1.charAt(
(0))--taC



COMPARISON OF STRINGBUFFER

COMPARISON OF STRINGBUFFER :

- **== ------------------------->**Compares the reference.
- **equals() ----------------->**Compares reference of the object.

DISADVANTAGE OF STRINGBUFFER

DISADVANTAGES :

Multiple thread can't execute the StringBuffer object simultaneously because all the methods are synchronized. So, Execution time is more. In order to overcome this problem we will go for String Builder.

NOTE :

The characteristics of StringBuffer and StringBuilder are same

DIFFERENCE BETWEEN STRINGBUFFER AND STRINGBUILDER

STRING BUFFER	STRING BUILDER
All the method present in StringBuffer is synchronized.	All the method present in StringBuilder is non synchronized.
At a time only one thread is allowed to access String Buffer object. Hence it is Thread safe.	At a time multiple thread is allowed to access String Builder object. Hence it is Not Thread safe.
Threads are required to wait to operate a stringBuffer object. Hence Relatively performance is low .	Threads are not required to wait to operate a StringBuilder object. Hence Relatively performance is high.
Less efficient than StringBuilder	Efficiency is high compared to StringBuffer
Introduced in 1.0 v	Introduced in 1.5v

ABSTRACTION

ABSTRACTION :

It is a design process of hiding the implementation and showing only the functionality (only declaration) to the user is known as abstraction.

HOW TO ACHIEVE ABSTRACTION IN JAVA ?

- In java we can achieve abstraction with the help of abstract classes and interfaces.
- We can provide implementation to the abstract component with the help of inheritance and method overriding.

ABSTRACT MODIFIER

ABSTRACT MODIFIER :

- Abstract is a modifier, it is a keyword.
- It is applicable for methods and classes.

ABSTRACT METHOD :

- A method that is prefixed with an abstract modifier is known as the abstract method.
- This is also said to be an incomplete method.
- The abstract method doesn’t have a body (it has the only declaration)

Syntax to create abstract method :

abstract[access modifier] [modifier] returnType methodName([For_Arg]) ;

NOTE :

Only child class of that class is responsible for giving implementation to the abstract method.

ABSTRACT CLASS

ABSTRACT CLASS :

If the class is prefixed with an abstract modifier then it is known as abstract class.

We can’t create the object (INSTANCE) for an abstract class.

NOTE :

- **We can’t instantiate an abstract class**
- **We can have an abstract class without an abstract method.**
- **An abstract class can have both abstract and concrete method.**
- **If a class has at least one abstract method either declared or inherited but not overridden it is mandatory to make that class as abstract class.**

EXAMPLE :

```
abstract class Atm
{
        abstract public double withdrawal();
        abstract public void getBalance();
        abstract public void deposit();
}
```

// hiding implementation by providing only functionality

Atm a = new Atm() // CTE

NOTE :

Only subclass of Atm is responsible for giving implementation to the methods declared in a Atm class.

CONCRETE CLASS

CONCRETE CLASS :

- If the class which is not prefixed with an abstract modifier then it is known as concrete class.
- In java, we can create an object (INSTANCE) only for concrete class.

IMPLEMENTATION OF ABSTRACT METHOD

Implementation of abstract method :

- If a class extend abstract class then it should give implementation to all the abstract method of the superclass.
- If inheriting class doesn’t like to give implementation to the abstract method of superclass then it is mandatory to make subclass as an abstract class.
- If a subclass is also becoming an abstract class then the next level child class is responsible to give implementation to the abstract methods.

STEPS TO IMPLEMENT ABSTRACT METHOD :

STEP 1:

Create a class.

STEP 2:

Inherit the abstract class/ component.

STEP 3:

Override the abstract method inherited (Provide implementation to the inherited abstract method).

EXAMPLE :

```
abstract class WhatsApp
{
        abstract public void send();
}
class Application extends WhatsApp
{
        public void send()
        {
                System.out.println(“Send() method is implemented”);
        }
}
```

Creating object and calling the abstract method :

```
WhatsApp w = new Application();
w.send() // Send() method is implemented
```

CONCRETE CLASS :

The class which is not prefixed with an abstract modifier and doesn’t have any abstract method, either declared or inherited is known as concrete class.

NOTE :

In java we can create object only for the concrete class.

CONCRETE METHOD :

The method which give implementation to the abstract method is known as concrete method.

INTERFACES

INTERFACES :

It ia a component in java which is used to achieve 100 % abstraction with multiple implementation.

Syntax to create an interface

```
[Access Modifier] interface InterfaceName
{
    // declare members
}
```

When an interface is compiled we get a class file with extension.class only

EXAMPLE :

```
interface Demo
{

}
```

Demo is an interface

Before

```
interface Demo
{

}
```

Demo.java

After

```
abstract interface Demo
{

}
```

Demo.class

```
Interface Demo1
{
            Int a; //CTE : Variable a is by default public, static, final. A final variable must be initialised.

}
```

```
 Interface Demo2
{
            Public void test()// CTE
            {

      }
}
```

```
Interface Demo3
{
            Static void main(String[] a)
            {
                        System.out.println("Hello World..!!!");
      }
}
```

What all are the members the can be declared in an interface ?

MEMBERS	CLASS	INTERFACE
Static variables	Yes	Yes, but only final static variables
Non static variables	Yes	No
Static methods	Yes	Yes, From JDK 1.8 v. NOTE : They are by default public in nature
Non static methods	Yes	Yes but we can have only abstract non static methods NOTE : Non static methods are by default ● public ● abstract
Constructors	Yes	No
Initializers (Static & non static block)	Yes	No

NOTE :

In interface all the members are by default have public access modifier

Why do we need an interface ?

- To achieve 100% abstraction.Concrete non static methods are not allowed.
- To achieve multiple inheritance.

What all the members are not inherited from an interface ?

Only static methods of an interface is not inherited to both class and Interface.

INHERITANCE

INHERITANCE WITH RESPECT TO INTERFACE :

An Interface can inherit any number of interfaces with the help of extend keyword.

EXAMPLE :

```
interface I1
{
}
interface I2 extends I1
{
}
```

NOTE :

The interface which is inheriting an interface should not give implementation to the abstract methods.

EXAMPLE 1 :

- Interface I1 have 3 methods

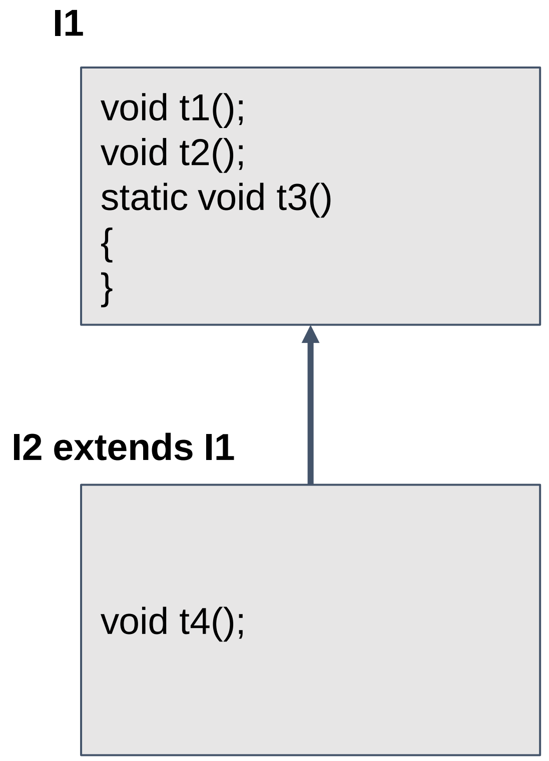
  2 - non static (**t1() , t2()**)

  1 - static (**t3()**)

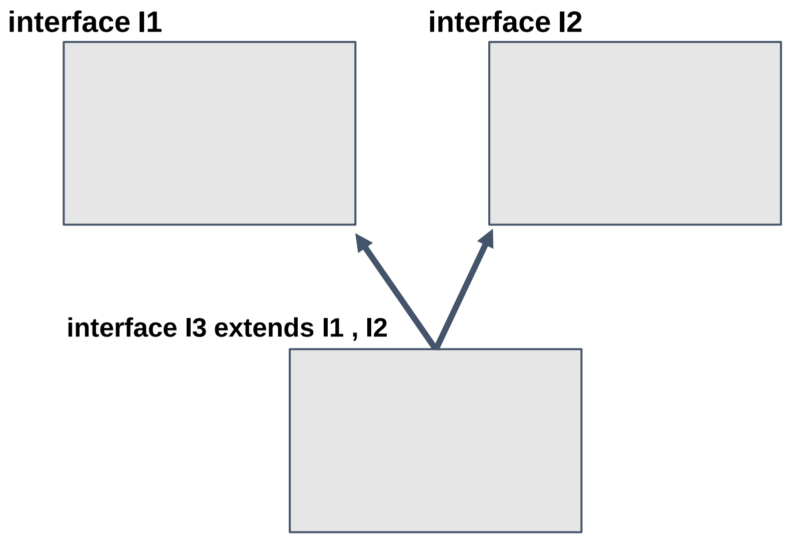
- Interface I2 have 3 methods

  2 - inherited non static method (**t1() , t2()**)

  1 - declared non static methods (**t4()**)



EXAMPLE 2 : Interface can inherit multiple interfaces at a time



NOTE :

With respect to interface there is no diamond problem.

Reason ,

- They don't have a constructors.
- Non static methods are abstract (do not have implementation)
- Static methods are not inherited.

INHERITANCE OF AN INTERFACE BY THE CLASS :

- Class can inherit an interface with the help of implements keyword.
- Class can inherit more than one interface.
- Class can inherit a class and an interface at a time.

NOTE :

- **If a class inherit an interface then it should give implementation to the abstract non static methods of an interface.**
- **If the class is not ready to give implementation to the abstract non static methods of an interface then it is mandatory to make that class as abstract method.**
- **Next level of child class is responsible for giving implementation to the rest of abstract non static methods of an interface.**

EXAMPLE 1 : Class inheriting an interface

- Interface I1 have 3 methods

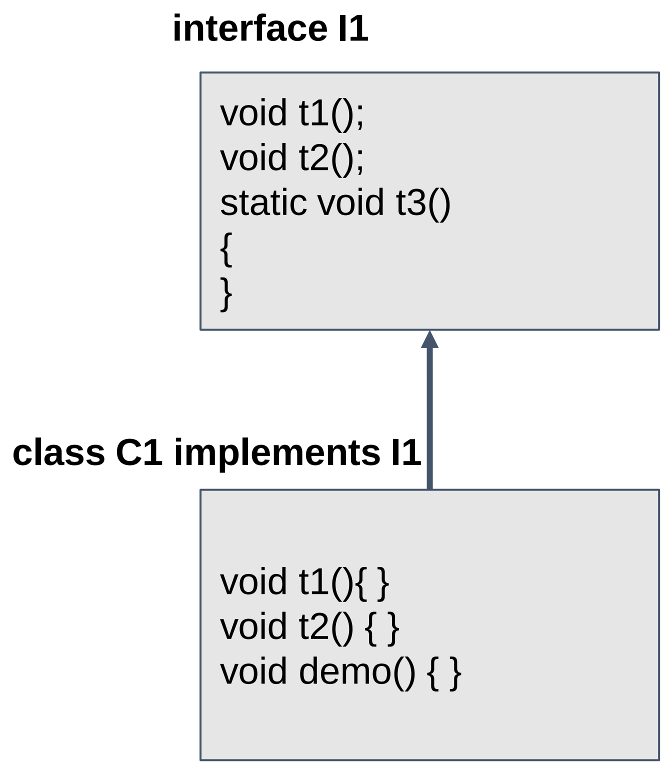
  2 - non static (**t1() , t2()**)

  1 - static (**t3()**)

- Class c1 have 3 methods

  2 - inherited and implemented non static method (**t1() , t2()**)

  1 - declared non static methods (**demo()**)

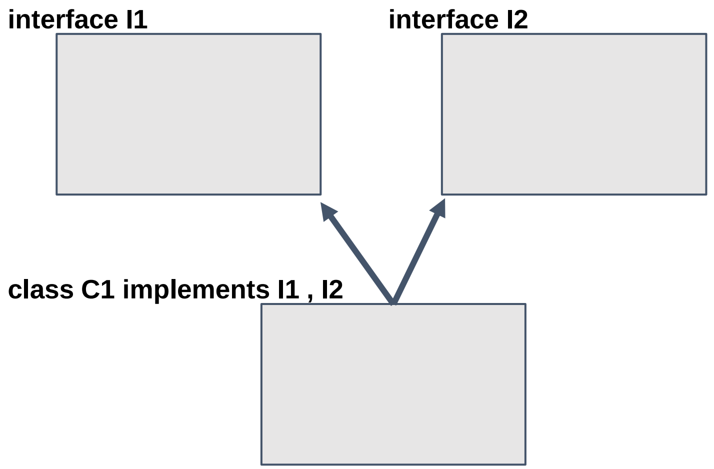


```
void t1();
void t2();
static void t3()
{
}
```

```
void t1(){ }
void t2() { }
void demo() { }
```

SOLUTION FOR DIAMOND PROBLEM

EXAMPLE 2 : Class inheriting multiple interfaces

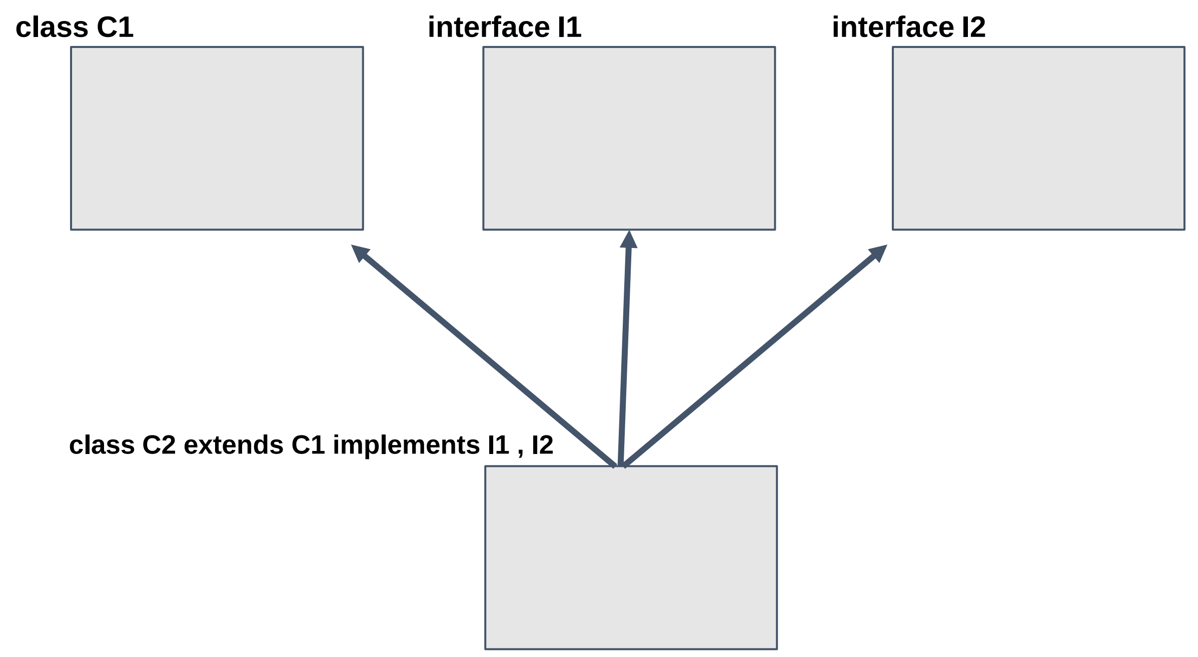


The diamond problem is solved by implementing multiple interface by the class at a time.

Reason,

- Non static methods are not implemented in an interface.
- Static methods are not inherited.

EXAMPLE 3 : Class can inherit interface and class at a time



RULE :

Use the extends first and then implements.

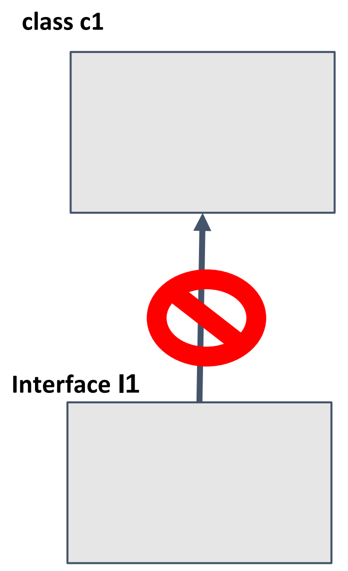
NOTE :

Class can inherit multiple interface but it can't inherit multiple class at a time.

Interface can't inherit a class.

NOTE

Interface can’t inherit from the class



Reason,

Class has concrete non static methods, So class can’t be a parent to the interface

PACKAGES

PACKAGES :

A package in java is used to group a related classes , interfaces and subclasses. In a simple word it is a folder / directory which consist of several classes and interfaces.

NOTE :

Package contains only class file

WHY PACKAGE ? :

- Packages are used to avoid name conflict.
- It increases maintainability.
- It is used to categorize classes and interfaces.
- It increases the access protection.
- It is used to achieve code reusability.

Types of packages :

We have two types of packages

- **Built in packages**
- **User defined packages**

BUILT IN PACKAGES

BUILT IN PACKAGES :

In java we have many built in packages like java, lang, util, io, sql, swing, awt, etc…Which are included in Java Development Kit.

EXAMPLE :

Java.lang.Math

java ---> package

lang ---> subpackage

Math ---> class

Subpackage :

Package inside a package is called as sub package.it should created to categorize a folder further.

HOW TO USE BUILT IN PACKAGES ?

We can use inbuilt packages by using two ways,

1. **By using fully qualified name.**
2. **By using import keyword.**

1.By using fully qualified name :

By using fully qualified name, compiler can understand to which package the specified class is available.

EXAMPLE :

There is one class named as ArrayList is available in java.util package. This class is used to group heterogeneous object as single entity. The fully qualified name for this class is java.util.ArrayList.

Creating object for ArrayList by using fully qualified name,

```
java.util.ArrayList al = new java.util.ArrayList();
```

By using fully qualified name, compiler can understand that ArrayList belongs to java.util Package.

DISADVANTAGE OF USING FULLY QUALIFIED NAME:

- We need use fully qualified name for every time when we accessing the class or interface.
- Readability is low.

To overcome this we can use a class by using import statement,

BY USING IMPORT STATEMENT :

- Import statement is used to import the classes or interfaces present in packages / Subpackages.

Syntax to use import statement:

import package.subpackage/class/interface ;

- By using import statement, instead of using fully qualified name for the classes we can directly use the class name.

RULE TO USE IMPORT STATEMENT:

1. **Import statement should be used before declaring a class.**
2. **import statement should be end with (;).**
3. **We can use multiple import statement in a same program.**

EXAMPLE:

```
import java.util.Scanner ;
class Demo
{
    Public static void main(String []args)
    {
            Scanner s = new Scanner(System.in);
    }
}
```

- By using this statement only Scanner class present inside the java.util package is get imported and accessible.
- If you want to use another class from java.util package use the import statement again before the class.

ADVANTAGE OF IMPORT STATEMENT :

- We can directly use class name instead of using fully qualified.
- By importing a package once we can use the class / interface of that package as many times possible.

NOTE :

java.lang package is implicitly imported by the compiler.

USER DEFINED PACKAGES

USER-DEFINED PACKAGE :

In java we can create our own package.

Syntax to create a package :

package package_Name;

Subpackage :

We can create subpackage by using following syntax.

Syntax to create package along with subpackage subpackage :

package package_name.subpackage_name ;

RULE :

- Package should be the first statement in a java program.
- A java source file should contain only one package Statement.
- A package can contain multiple classes/Interfaces but utmost one class/interface should be public.
- If you want to add more then one public class inside the same package then, create a separate source file for each public class with same package name.
- If a package contains public class/interface then it is mandatory to use public class/interface name as a source file name. Otherwise, we will get **Compile Time Error.**

EXAMPLE :

```
package mypack;
public class PackageDemo
{
public static void main(String[] args)
{
    System.out.println(“Package is created successfully”);
}
}
```

To compile : javac -d . PackageDemo.java

To execute : java mypack PackageDemo

NOTE :

If we want to use package and import statement in a same program then we should follow a sequence of program,

package

import

class / interface / enum

ACCESS MODIFIERS

We have two type of modifiers

1. Access modifiers
2. Non access modifiers

Access modifiers :

- Access modifier are responsible to change / modify the accessibility of the member.
- We have four type of access modifiers,

    1. **private**
    2. **default**
    3. **protected**
    4. **Public**

PRIVATE

PRIVATE ACCESS MODIFIER :

- It is a class level modifier , it is applicable for variables , method and constructors.
- If the member of a class is prefixed with private modifier then it is accessible only within the class accessing outside the class is not possible.

EXAMPLE :

```
class A{
private static int i ;
}
class B{
Public static void main(String[] args){
System.out.println(A.i); // CTE
}
```

DEFAULT

DEFAULT ACCESS MODIFIER :

- The accessibility of default modifier is only within the package.It can’t be accessed from outside the package.
- If you don’t declare any access modifier then it is considered as a default access modifier.

EXAMPLE :

```
package myPack ;
public class Demo{
        static int i ;
}
class Driver{
Public static void main(String[] args){
Demo.i = 5 ; // CTE
}
```

PROTECTED ACCESS MODIFIER :

- The access level of a protected modifier is within the package and outside the package through child class.
- If you do not make the child class, it cannot be accessed from outside the package.

PUBLIC ACCESS MODIFIER :

- The access level of a public modifier is anywhere.
- It can be accessed from within the class, outside the class, within the package as well as outside the package.

SCOPE OF AN ACCESS MODIFIER

ACCESS MODIFIER	WITHIN THE CLASS	WITHIN THE PACKAGE	OUTSIDE THE PACKAGE	OUTSIDE THE PACKAGE BY THE CHILD CLASS
private	YES	NO	NO	NO
default	YES	YES	NO	NO
protected	YES	YES	YES	NO
public	YES	YES	YES	YES

The scope of the access modifier based on the accessibility is :

private < default < protected < public

ARRAY

ARRAY :

Array is a continuous block of memory which is used to store multiple values.

CHARACTERISTIC OF AN ARRAY :

- The size of an array must be defined at the time of declaration.
- Once declared , the size of an array can't modified.
- Hence array is known as fixed size.
- In an array we can access the elements with the help of an index or subscript. It is an integer number that starts from 0 and ends at length of the array-1.
- In an array we can store only homogeneous type value. It is also known as homogeneous collection of an object.

NOTE :

In java array is an object

DECLARING AN ARRAY

Syntax to declare an array :

datatype[] variable; (or) datatype variable[]

EXAMPLE :

int a[] **---** single dimensional array reference variable of int type

float f[] **---** single dimensional array reference variable of float type.

String s[] **---** single dimensional array reference variable of String type.

INSTANTIATING AN ARRAY

Syntax to instantiate an array :

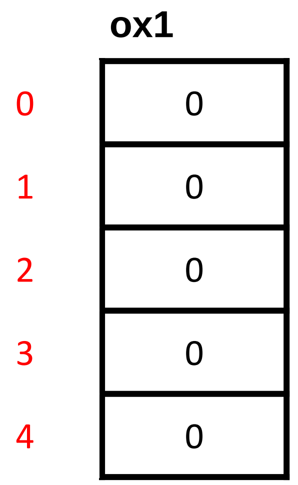
new datatype[size];

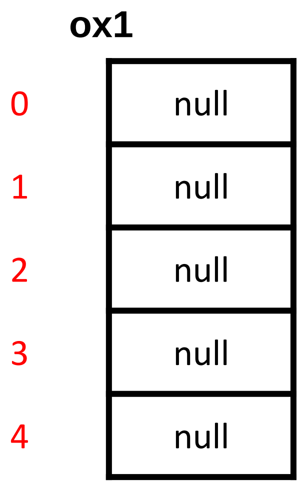
EXAMPLE :

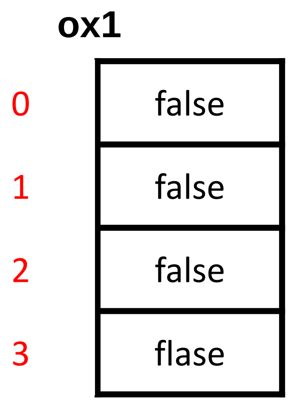
new int[5];

new boolean[4]

new String[5];







NOTE :

Once the array is instantiated it is assigned with default value

INITIALIZING AN ARRAY

ADDING ELEMENTS :

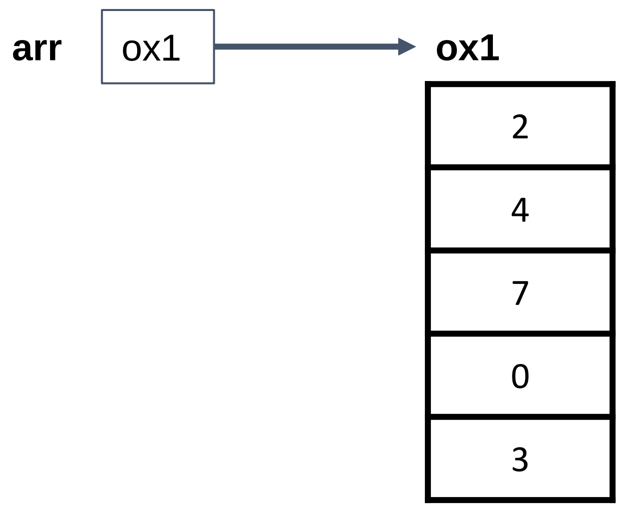
We can add an element into an array with the help of array index.

Syntax to add an element into an array :

array_ref_variable[index] = value ;

EXAMPLE :

```
int[] arr = new int[5] ; //declaration
arr[0] = 2;
arr[1] = 4;
arr[2] = 7;
arr[3] = 0;
arr[4] = 3;
```



ACCESSING ELEMENTS FROM AN ARRAY

ACCESSING ELEMENTS :

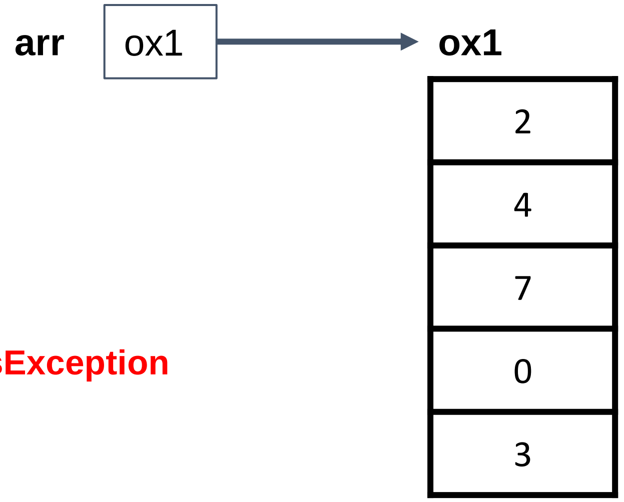
We can access an element from an array with the help of array reference variable and index.

Syntax to access an elements from an array :

array_ref_variable[index] ;

EXAMPLE :

```
System.out.println(arr[0]); // 2
System.out.println(arr[2]); // 7
System.out.println(arr[4]); // 3
System.out.println(arr[5]); //ArrayIndexOutOfBoundsException
```



Declaring , Instantiating , Initializing An Array In A Single Line

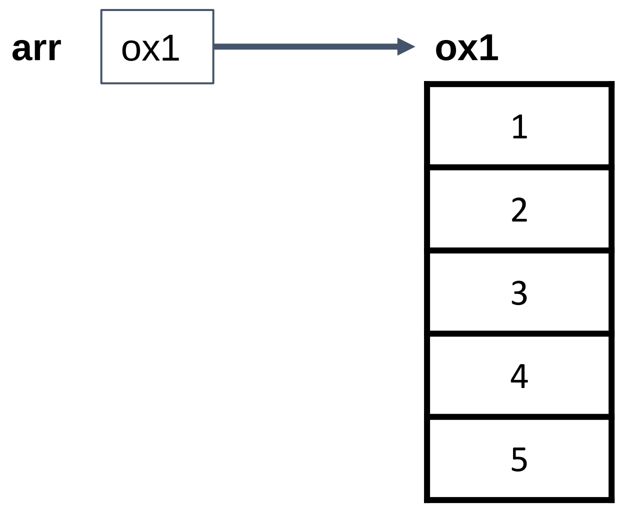
We can also declare , instantiate and initialize an array by using single line.

Syntax :

datatype[] arr_ref_var = { element1 , element2 , element3 , etc... } ;

EXAMPLE :

int arr[] = { 1 , 2 , 3 , 4 , 5 };



EXCEPTION

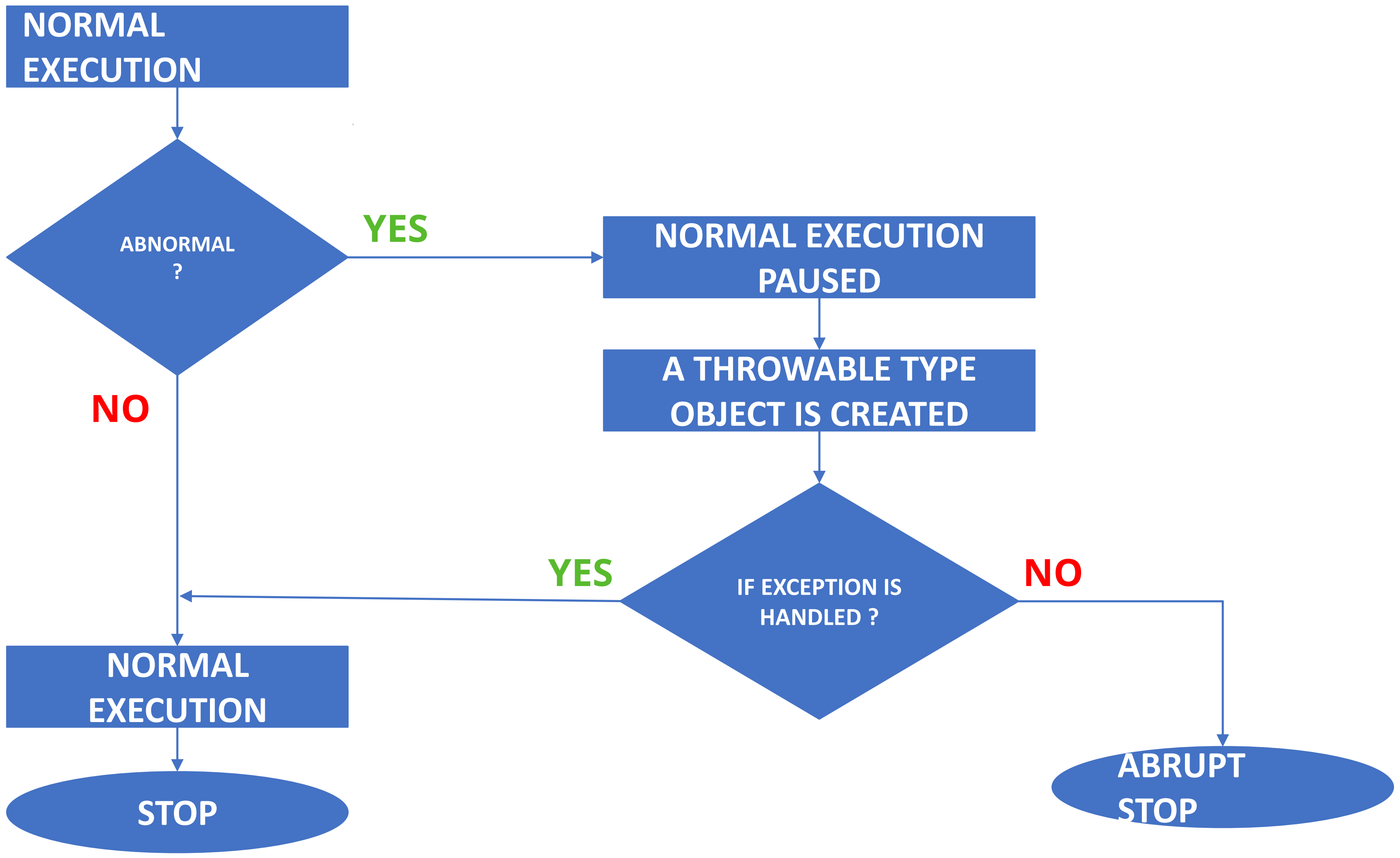
EXCEPTION :

The exception is a problem that occurs during the execution of a program (Runtime). When an exception occurs, the execution of the program stops abruptly (Unexpected stop).

NOTE :

Every exception in java is a class of 'Throwable type'

WHAT HAPPENS IF AN EXCEPTION OCCURS ?



NOTE :

- **Every exception is occurred because of a statement.**
- **A statement will throw an exception during the abnormal situation.**

EXAMPLE :

```
        import java.util.Scanner ;
    class
        {
                public static void main(String[] args)
                {
                        Scanner input = new Scanner(System.in);
                        int a , b ;
                        a = input.nextInt();
                        b = input.nextInt();
                        int c = a / b ; // Statement might cause
Exception(ArithmeticException)
                        System.out.println("The division of "+a+" and "+b+" = "+c);
                }
        }
```

CASE 1	CASE 2
a = 5 , b = 10 ; **->**Normal situation **->**No exception	**a = 5 , b = 0 ;** **->**Abnormal situation **->**Exception occurs

IMPORTANT EXCEPTIONS AND STATEMENTS

STATEMENT	EXCEPTION
a/b	ArithmeticException
reference.member	NullPointerException
(ClassName)reference	ClassCastException
array_ref[index]	ArrayIndexOutOfBoundsException
string.charAt(index)	StringIndexOutOfBoundsException
string.substring(Index)	StringIndexOutOfBoundsException

CHECKED EXCEPTION

CHECKED EXCEPTION :

The compiler-aware exception is known as the checked exception. i.e., the Compiler knows the statement responsible for abnormal situations (Exception). Therefore the compiler forces the programmer to either handle or declare the exception. If it is not done we will get an unreported compile-time error.

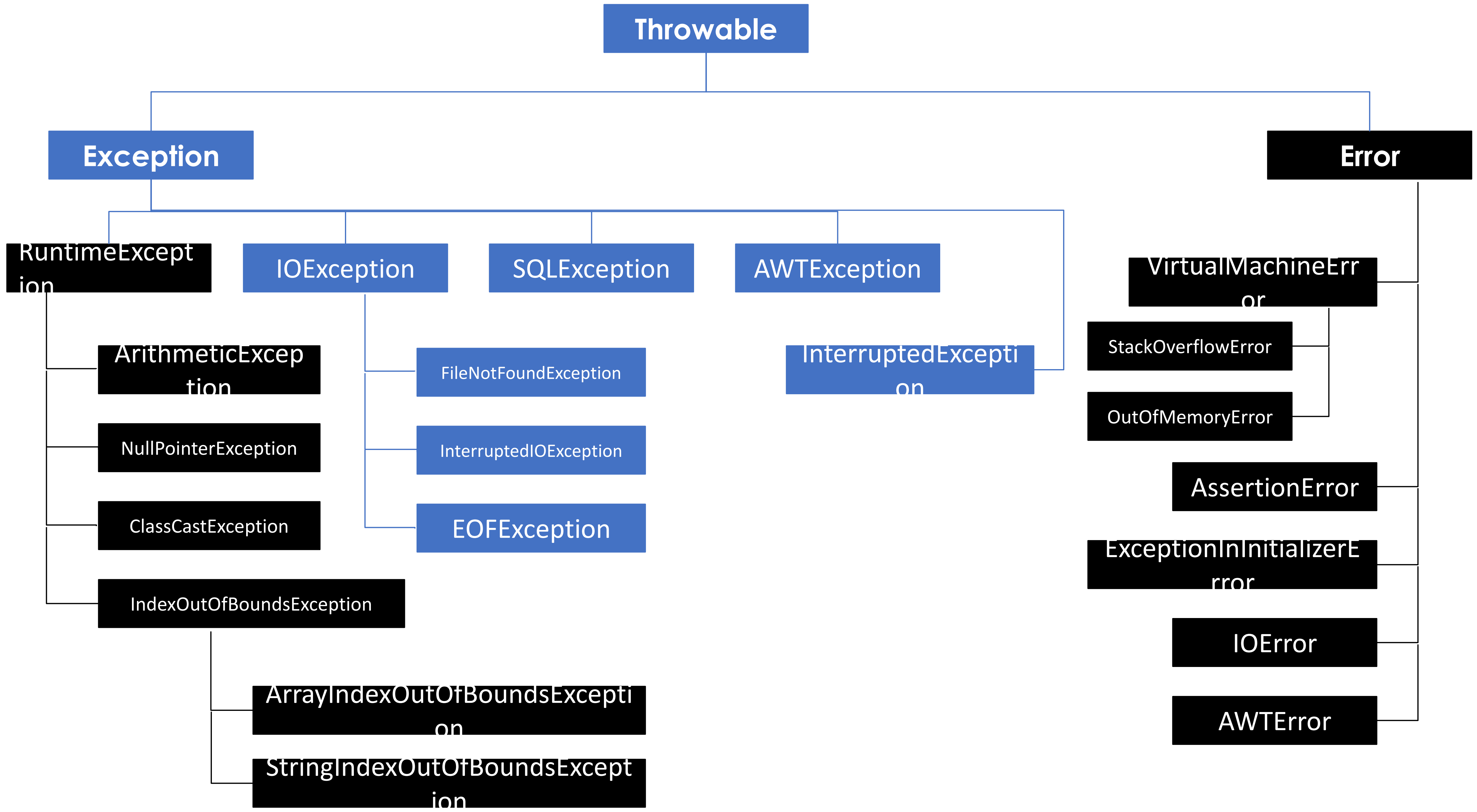
Example : FileNotFoundException

UNCHECKED EXCEPTION

UNCHECKED EXCEPTION :

The compiler-unaware exception is known as the unchecked exception. i.e., the Compiler doesn’t know the statements which are responsible for abnormal situations (Exception). Hence, the compiler will not force the programmer either to handle or declare the exception.

Example : ArithmeticException


Throwable
Exception
Error
RuntimeException
IOException
SQLException
AWTException
InterruptedException
ArithmeticException
NullPointerException
ClassCastException
IndexOutOfBoundsException
ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException
FileNotFoundException
InterruptedIOException
EOFException
VirtualMachineError
StackOverflowError
OutOfMemoryError
AssertionError
ExceptionInInitializerError
IOError
AWTError

NOTE :

- In Throwable hierarchy Error class and its subclasses, RuntimeException class and its subclasses are all considered as Unchecked Exceptions.
- All the subclasses of the Exception class except RuntimeException are considered as Checked Exceptions.
- Throwable and Exception classes are partially checked and partially unchecked.

THROWABLE

Throwable :

Throwable class is defined in java.lang package.

NOTE :

In the Throwable class, all the built-in classes of the Throwable type are overridden toString() method such that it returns the fully qualified name of the class.

Important methods of Throwable class :

- String getMessage();
- void printStackTrace();

EXCEPTION HANDLING

EXCEPTION HANDLING :

Exception handling is a mechanism used in java that is used to continue the normal flow of execution when the exception occurred during runtime.

HOW TO HANDLE THE EXCEPTION?

In java, we can handle the exception with the help of a try-catch block.

Syntax to use try catch block :

```
try
{
        // Statements ;
}
catch(declare one variable of Throwable type)
{
        // Statements ;
}
```

try-catch BLOCK

try{ }:

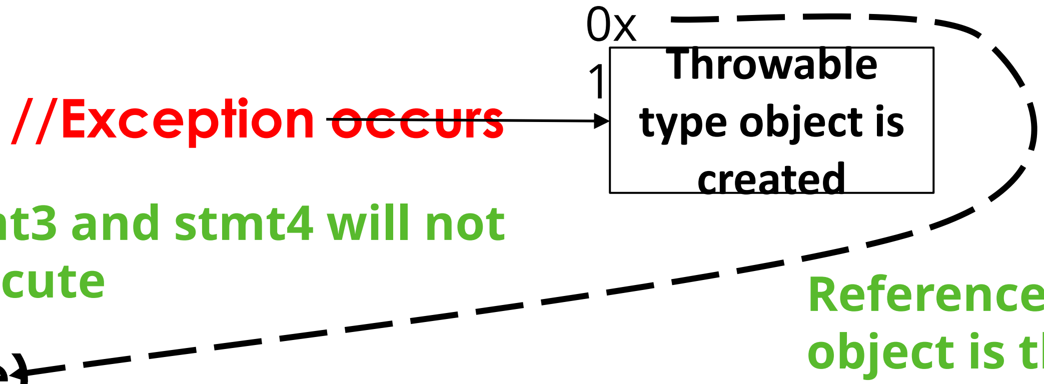
- The statements which are responsible for exceptions should be written inside the try block.
- When an exception occurs,

  I. **Execution of try block is stopped.**
  II. **A Throwable type object is created.**
  III. **The reference of throwable type object created is passed to the catch block.**

EXAMPLE :

```
        try
        {
                stmt1;
                stmt2;   //Exception occurs
                stmt2;
                stmt3;
        }
        catch(variable)
        {

        }
```



stmt3 and stmt4 will not execute

Reference of Throwable type object is thrown to the catch block

catch() { } :

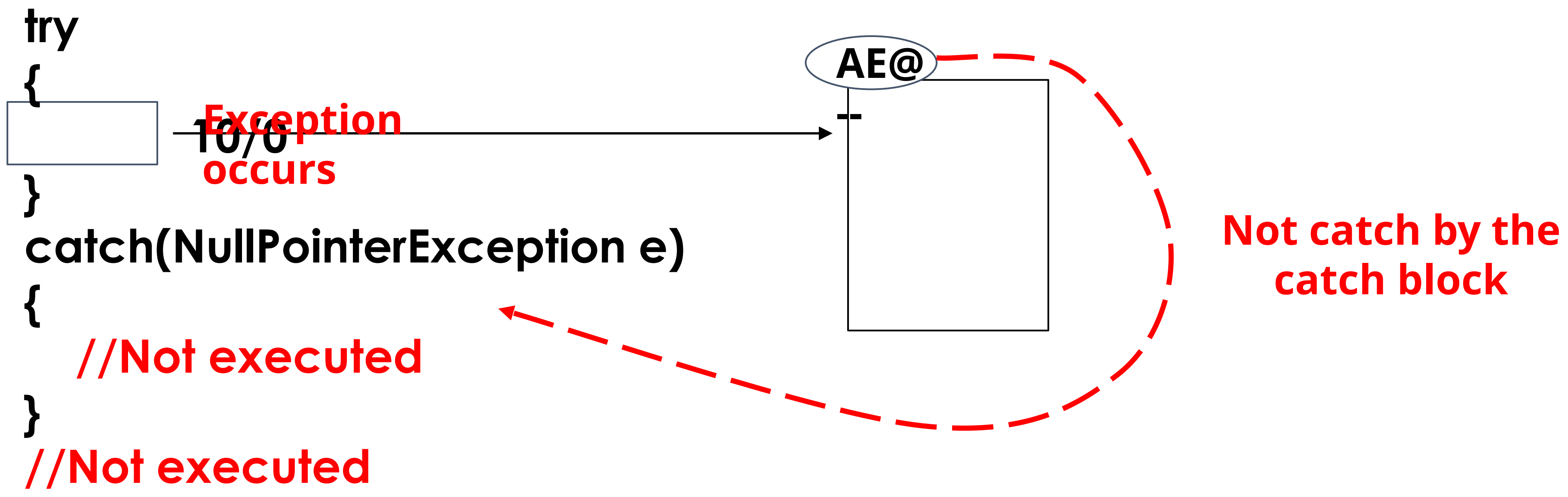
The catch block is used to catch the throwable type reference thrown by the try block.

1. If it catches, We say the exception is handled. Statements inside the catch block are get executed and the normal flow of the program will continue.
2. If it doesn't catch, we say the exception is not handled. Statements written inside the catch block are not executed and the program is terminated.

When do we say exception is handled ?

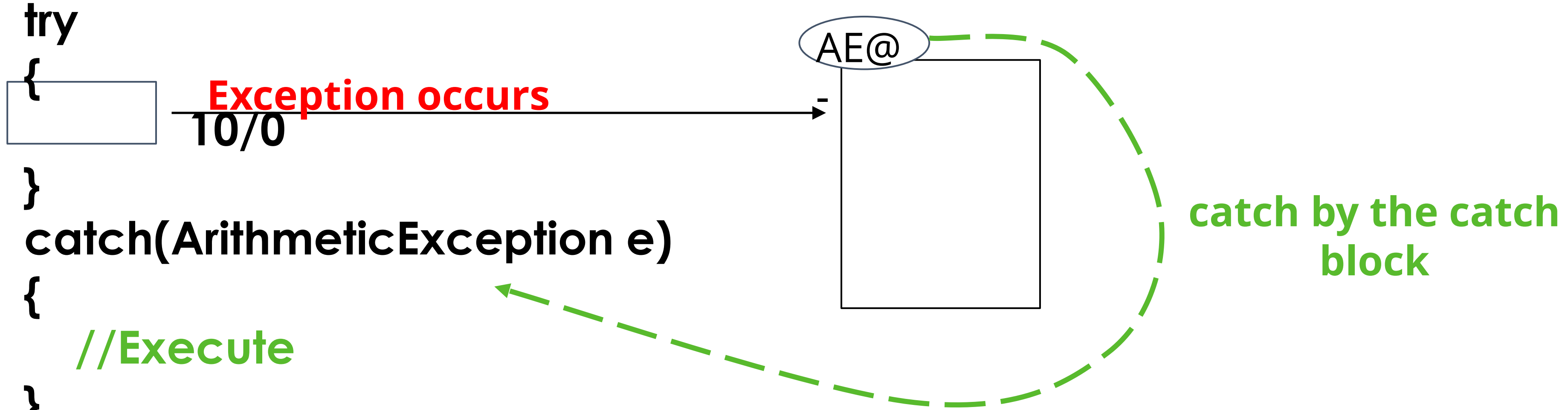
We say Exception is handled only if Exception is caught by catch block.

Case 1 : Exception occurs not caught :



Case 2 : Exception occurs and caught :

```
try
{
    10/0
}
catch(ArithmeticException e)
{
    //Execute
}
//Execute
```



EXAMPLE :

EXCEPTION OCCURS BUT NOT HANDLED

```
System.out.println(“Main Begins”);
try()
{
   int c = 5/0;
}
catch(NullPointerException e)
{
System.out.println(“Divisible by zero is not
possible”);
}
System.out.println(“Main End”);
```

CONSOLE :

```
   Main Begins
   Exception in thread "main"
java.lang.ArithmeticException: / by zero
```

EXCEPTION OCCURS AND HANDLED

```
System.out.println(“Main Begins”);
try()
{
   int c = 5/0;
}
catch(ArithmeticException e)
{
System.out.println(“Divisible by zero is not
possible”);
}
System.out.println(“Main End”);
```

CONSOLE :

```
   Main Begins
   Divisible by zero is not possible
   Main End
```

TRY WITH MULTIPLE CATCH BLOCK

try with multiple catch :

try block can be associated with more than one catch blocks

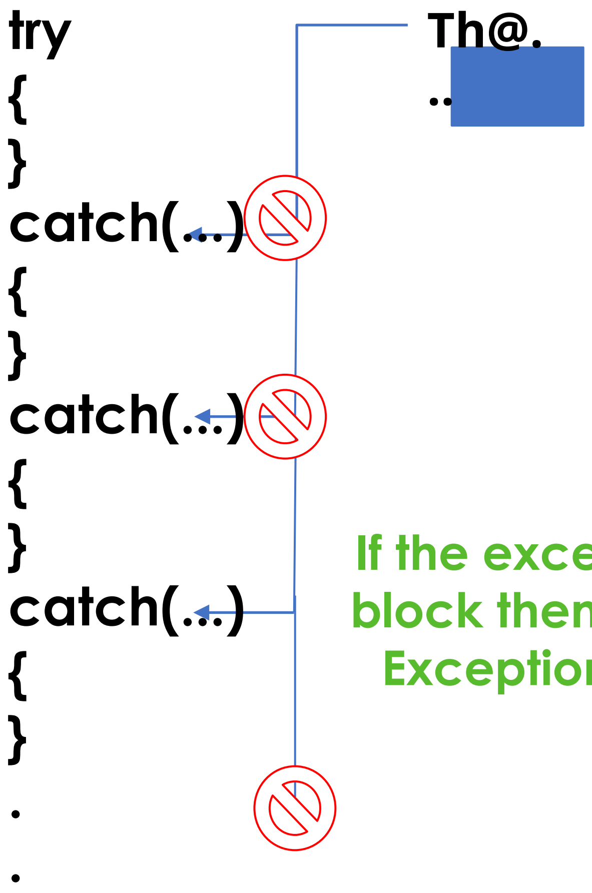
Syntax :

```
try
{
}
catch(...)
{
}
catch(...)
{
}
.
.
.
```

NOTE :

The Exception type object thrown from top to bottom order.

WORKFLOW :



```
try
{
}
catch(...)
{
}
catch(...)
{
}
catch(...)
{
}
.
.
```

If the exception is caught by the catch block then try block does not throw the Exception object to the below catch blocks

RULE :

The order of catch block should be maintained such that, child type should be on the top and parent type at the bottom.

EXAMPLE :

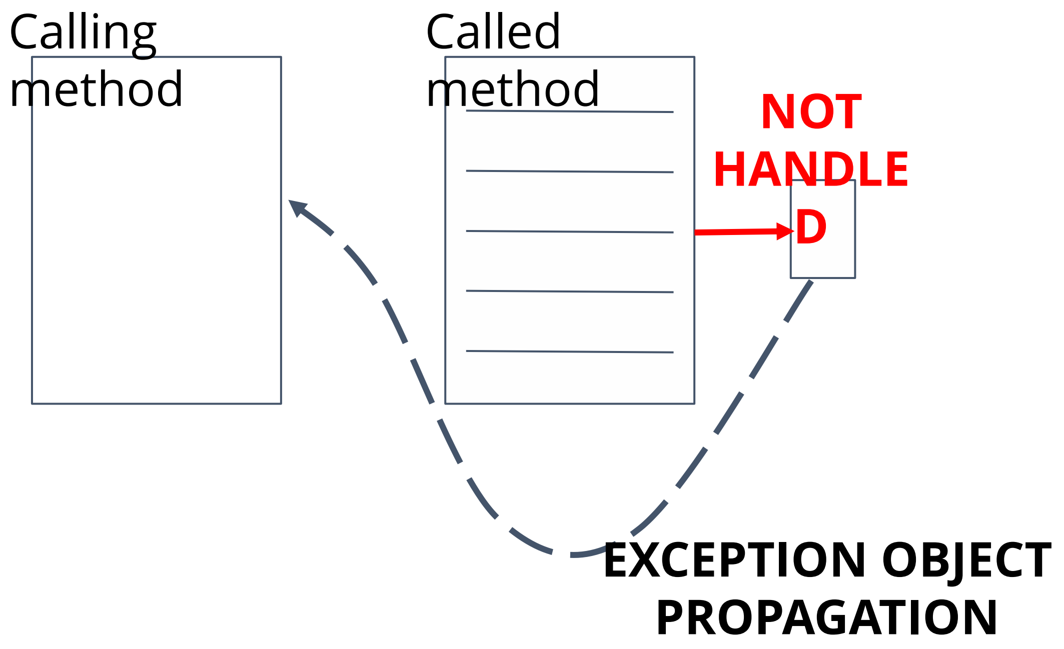
CASE 1	CASE 2

try
{
        10/0;
}
catch(Exception e)
{
}
catch(ArithmeticException e)
{
}
 	
CTE : Parent type is declared on the top and child type is on bottom
	try
	{
	        10/0;
	}
	catch(ArithmeticException e)
	{
	}
	catch(Exception e)
	{
	}
	
	CTS : Parent type is declared on the bottom and child type is on top

EXCEPTION OBJECT PROPAGATION

EXCEPTION OBJECT PROPAGATION :

The movement of exception from called method to calling method when it is not handled is known as Exception object propagation.

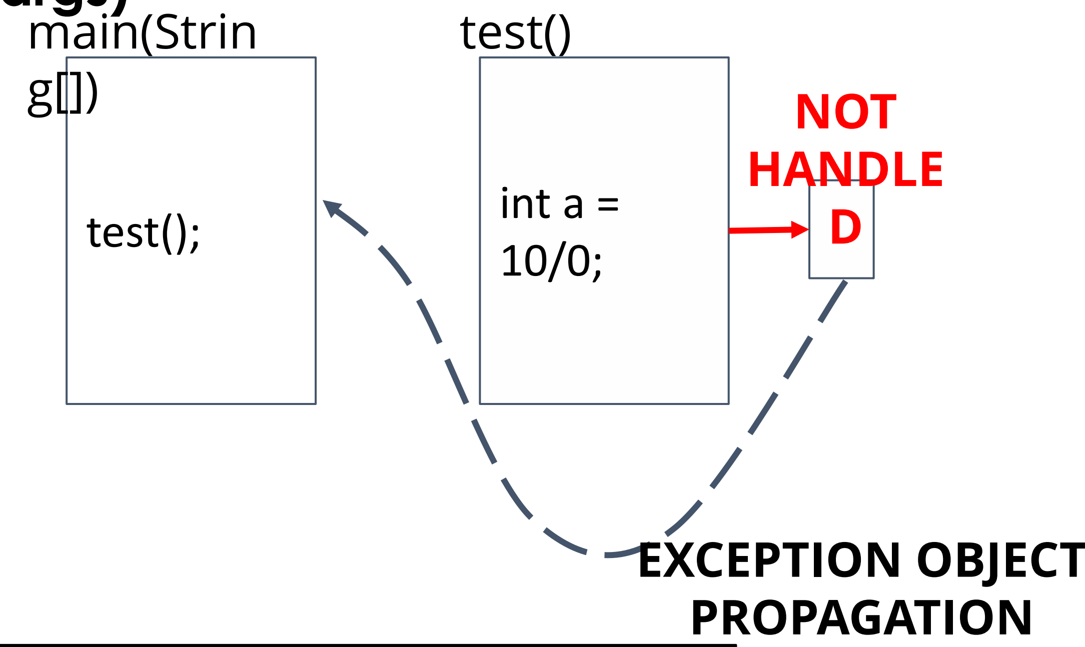


CASE : EXCEPTION OCCURRED AND HANDLED BY THE CALLING METHOD :

```
        class Case1
        {
                public static void main(String[] args)
                {
                        try
                        {
                                test();
            }
            catch(ArithmeticException e)
                        {
                                System.out.println("Exception is handled by the
calling method");
                        }
                }
                static void test()
                {
                        Int a = 10/0;
                }
        }
```

CASE : EXCEPTION OCCURRED AND NOT HANDLED BY THE CALLING METHOD:

```
class Case1
{
        public static void main(String[] args)
        {
                test();
        }
        static void test()
        {
                Int a = 10/0;
        }
}
```



```
Exception in thread "main" java.lang.ArithmeticException: / byzero
        at Case1.test(Case1.java: 8)
        at Case1.main(Case1.java: 4)
```

CHECKED EXCEPTION

CHECKED EXCEPTION :

- The compiler-aware exception is known as checked exception.
- If any statements are responsible for checked exception it is mandatory to either declare or handle the exception otherwise, we will get unreported compile time error.
- Compiler will force the programmer to either declare or handle the checked exception during compile time.

Example: FileNotFoundException

FileNotFoundException :

- FileNotFoundException is defined in the java.io package.
- We get FileNotFoundException when we try to create a file but the given path is wrong or no permission or there is not sufficient memory in the hard disk.
- **new FileOutputStream("path/name")** ---> this statement is responsible for
- FileNotFoundException.
- FileNotFoundException is a **checked exception.**

EXAMPLE 1:

```
        import java.io.FileOutputStream;
    class FileNotFoundDemo1
        {
                public static void main(String[] args)
                {
                        // to create a file demo.txt in e://f1
                        FileOutputStream fout = new
FileOutputStream(“e://f1/demo.txt”);
                        System.out.println(“File is created”);
                }
        }
```

We will get CTE, because new FileOutputStream(“e://f1/demo.txt”) is responsible for FileNotFoundException (Checked Exception), it is neither declared nor handled.

EXAMPLE 2: Resolving Compile Time Error by handling checked exception

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
class FileNotFoundDemo2
    {
        public static void main(String[] args)
        {
            // to create a file demo.txt in e://f1
            try
            {
                FileOutputStream fout = new
FileOutputStream("e://f1/demo.txt");
                System.out.println("File is created");
            }
            catch(FileNotFoundException e)
            {
                System.out.prinltn(e);
            }
            System.out.printn("Main end");
        }
    }
```

THROW KEYWORD

throw :

- It is a keyword.
- It is used to throw an exception manually.
- By using throw we can throw checked exception, unchecked exception. It is mainly used to throw the custom exception.

SYNTAX :

```
throw exception ;
throw new CustomException("String");
```

EXAMPLE :

```
        class throwDemo
        {
                public static void main(String[] args)
                {
            int a = 15;
            int b = 10;
            if(a>b)
                throw new ArithmeticException(“manually thrown”);
            else
                System.out.println(“No exception”);
                }
        }
```

OUTPUT :

```
Exception in thread main java.lang.ArithmeticException: manually thrown
```

NOTE :

If we don’t handle the manually thrown exception then we will get unreported exception.

CUSTOM EXCEPTION :

```
class NotValidException extends Exception
{
        NotValidException(String s)
        {
                super(s);
        }
}
```

Raising and handing the NotValidException

```
class Driver
{
        public static void main(String[] args)
        {
                try
                {
                        throw new NotValidException();
                }
                catch(NotValidException e)
                {
                        System.out.println("Custom exception is handled");
                }
        }
}
```

THROWS KEYWORD

throws :

- It is a keyword.
- It is used to declare an exception to be thrown to the caller.

NOTE :

throws keyword should be used in the method declaration statement.

SYNTAX :

```
        [modifier] return_type methodName([formal arg]) throws excep1,
excep2,...
        {
                //stmts;
        }
```

NOTE :

If a method declares an exception using throws keyword, then caller of the method must handle or throw the exception. If not we will get compile time error.

EX

DECLARATION	PURPOSE
public static void sleep(long) throws InterruptedException	The purpose of this method is to pause the execution of a program to the specified time.

EXAMPLE 2: InterruptedException is not either handled or declared

```
class throwsDemo1
{
        public static void main(String[] args)
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(“Hello”);
                        Thread.sleep(5000);
                }
        }
}
```

We get CTE, because Thread.sleep(long) method declares InterruptedException (Checked Exception). main(String[] args) is a caller of Thread.sleep(long) method but it is not declaring or handling the InterruptedException.

EXAMPLE 2: InterruptedException handled by main method

```
class throwsDemo2
{
        public static void main(String[] args)
        {
                for(int i=1;i<=5;i++)
                {
                        System.out.println(“Hello”);
                        //
                        Try

                        {
                                Thread.sleep(5000);
                        }
                        catch(InterruptedException e)
                        {
                                System.out.println(e);
                        }
                }
        }
}
```

EXAMPLE 2: InterruptedException instead of handling again thrown by main method

```
class throwsDemo2
{
	public static void main(String[] args) throws InterruptedException
	{
		for(int i=1;i<=5;i++)
		{
			System.out.println(“Hello”);
			Thread.sleep(5000);
		}
	}
}
```

NOTE :

throws keyword does not handle the exception

FINALLY BLOCK

finally { }

- It is a block which is used along with try-catch block.
- It will execute even the exception is not handled

Syntax :

```
try
{
}
catch(...)
{
}
finally
{
}
```

NOTE :

We can also use finally block with try block alone.

FINAL VS FINALLY

final	finally{ }
final is a modifier	finally is a block
It is applicable to class, method, variable. final class can’t be inherited, final variable value can’t be changed , final method can’t be inherited	Instruction written inside the ‘finally block’ will be executed even if the exception isn’t handled.

WRAPPER CLASS

WRAPPER CLASS :

- The wrapper class in java provides mechanism to wrap the primitive into an object.
- For every primitive data type corresponding class is declared known as a wrapper class.
- There are eight wrapper classes declared in java.lang package which provides several methods to convert primitive into an object.

NOTE :

Since J2SE 5.0 Conversion of primitive to Object and vice versa is implicitly done by the compiler.

WRAPPER CLASSES

PRIMITIVE DATA TYPES	WRAPPER CLASSES
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

NOT

Among these wrapper classes Byte, Short, Integer, Long, Float, Double are subclasses of Number class.

AUTOBOXING

AUTOBOXING :

The process of implicitly converting a primitive data type into corresponding wrapper class(Object) is known as autoboxing.

EXAMPLE :

int-Integer, byte-Byte, boolean-Boolean, etc.,

EXAMPLE :Autoboxing of int to Integer type

```
class Case1
{
	public static void main(String[] args)
	{
			int i = 10;
			Integer Obj = i ; //Autoboxing
	}
}
```

AUTO UNBOXING

AUTO UNBOXING :

The process of implicitly converting a object into primitive data type is known as auto unboxing.

EXAMPLE

Integer-int, Byte-byte, Boolean- boolean, etc.,

EXAMPLE: AutoUnboxing of Integer to int type

```
class Case1
{
	public static void main(String[] args)
	{
			Integer obj = 10;
			int i = obj ; //Auto Unboxing
	}
}
```

valueOf() METHOD

valueOf() Method :

We can wrap a primitive value to corresponding Wrapper class object by using a valueOf() method.

Declaration :

```
public static wrapper Value valueOf(String);
public static wrapper Value valueOf(primitive data);
```

EXAMPLE :

```
class Demo
{
        Public static void main(String[] args)
        {
                        //primitive data
                        byte b = 10;
                        short s = 20;
                int i = 30;
                long l = 40;
                float f = 10.0f;
                double d = 20.05;
                char c = ‘a’;
                        //converting primitive to object
                        Byte obj1 = Byte.valueOf(b);
                        Short obj2 = Short.valueOf((s);
                        Integer obj3 = Integer.valueOf(i);
                        Long obj4 = Long.valueOf(l);
                        Float obj5 = Float.valueOf(f);
                        Double obj6 = Double.valueOf(d);
                        Character obj7 = Character.valueOf(c); //CTE
                        String str = String.valueOf(i);
        }
}
```

[primitiveDataType]Value() METHOD

primitiveDataType**Value() Method :**

This method is used to find the primitive value for given Wrapper Object.

Declaration :

```
    public byte byteValue();
public short shortValue();
public int intValue();
public long longValue();
public float floatValue();
public double doubleValue();
```

NOTE :

- **It is declared inside all the six sub classes of a Number class(Byte, Short, Int, Long, Float, Doble).**
- **Additional to this Character class has charValue() method and Boolean class has booleanValue() method.**

EXAMPLE 1:

```
class Demo1
{
        public static void main(String[] args)
{
        // Object
        Long i = 20l;
        // Converting Object to primitive value
        byte b = i.byteValue();
        short s = i.shortValue();
        int in = i.intValuelong();
    long l = i.longValue();
        float f = i.floatValue();
        double d = i.doubleValue();
}
}
```

EXAMPLE 2:

```
class Demo2
{
        public static void main(String[] args)
{
        // Object
        Character obj = 'c';
        // Converting Object to primitive value
        char ch = obj.charValue();


}
}
```

EXAMPLE 3:

```
class Demo3
{
        public static void main(String[] args)
{
        // Object
        Boolean obj = false;
        // Converting Object to primitive value
        boolean b = obj.booleanValue();


}
}
```

parse[PrimitiveDataType]() METHOD

parse[PrimitiveDataType]**() Method :**

This method is used to convert a given string into a primitive value except the character.

Declaration :

Byte	: public static byte parseByte(String) throws NumberFormatException;
Short	: public static short parseShort(String) throws NumberFormatException;
Integer	: public static int parseInt(String) throws NumberFormatException;
Long	: public static long parseLong(String) throws NumberFormatException;
Float	: public static float parseFloat(String) throws NumberFormatException;
Double	: public static double parseDouble(String) throws NumberFormatException;
Boolean	: public static boolean parseBoolean(String)

NOTE :

The runtime string should be of number format, otherwise we will get NumberFormatException during runtime.

EXAMPLE 1:

```
class Demo
{
    public static void main(String[] args)
    {
            String str = "10";
            byte b = Byte.parseByte(str);
            short s = Short.parseShort(str);
            int i = Integer.parseInt(str);
            float f = Float.parseFloat(str);
            double d = Double.parseDouble(str);
    }
}
```

EXAMPLE 2: Raising NumberFormatException

```
class Demo
{
    public static void main(String[] args)
    {
            String str = "ten";
            byte b = Byte.parseByte(str);
            short s = Short.parseShort(str);
            int i = Integer.parseInt(str);
            float f = Float.parseFloat(str);
            double d = Double.parseDouble(str);
    }
}
```

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "ten"
        at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Byte.parseByte(Byte.java:149)
        at java.lang.Byte.parseByte(Byte.java:175)
```

EXAMPLE : public static boolean parseBoolean(String);

```
class Demo
{
	public static void main(String[] args)
	{
			String s = "True";
			boolean b = Boolean.parseBoolean(s);
			System.out.println(b); //true
	}
}
```

NOTE :

If any data passed other than a boolean in a parseBoolean() as a String, then the method will return a false.

COLLECTION FRAMEWORK

Why do we need collection Framework in java ?

To store multiple objects or group of objects together we can generally use arrays.But arrays has some limitations,

LIMITATION OF AN ARRAY

Limitations of an Array :

1. The size of the array is fixed, we cannot reduce or increase dynamically during the execution of the program.
2. Array is a collection of homogeneous elements.
3. Array manipulation such as :
   a. **removing an element from an array.**
   b. **adding the element in between the array etc...**

Requires complex logic to solve.

Therefore, to avoid the limitations of the array we can store the group of objects or elements using different data structures such as :

1. **List**
2. **Set**
3. **Queue**
4. **maps / dictionaries**

Def : Collection Framework is set of classes and interfaces (hierarchies), which provides mechanism to store group of objects (elements) together.

CRUD OPERATION

It also provides mechanism to perform actions such as :

1. create and add an element
2. access the elements
3. remove / delete the elements
4. search elements
5. update elements
6. sort the elements

(CRUD - operations)

Important hierarchies of collection framework

Collection framework has two important hierarchy,

1. **Collection hierarchy**
2. **Map hierarchy**

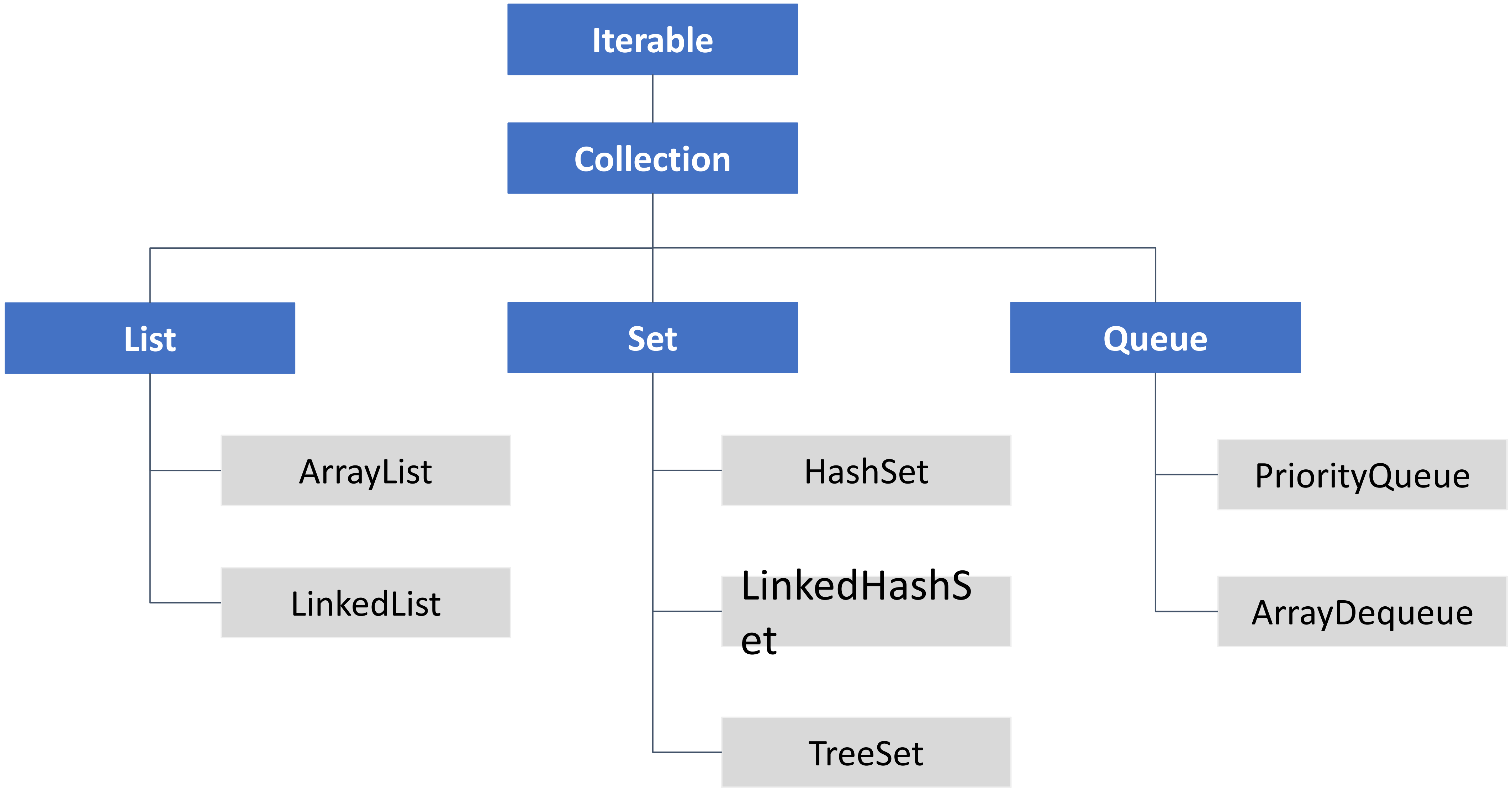
Collection interface

Collection interface :

1. Collection is an interface defined in java.util.
2. Collection interface provides the mechanism to store group of objects(elements) together.
1. All the elements in the collection are stored in the form of Objects. (i.e.only Non-primitive is allowed)
2. which helps the programmer to perform the following task.

a. add an element into the collection
b. search an element in the collection
c. remove an element from the collection
d. access the elements present in the collection

Collection hierarchy



IMPORTANT METHODS OF COLLECTION INTERFACE :

PURPOSE	RETURN TYPE	METHODS
To add an element	boolean	add(Object)
		addAll(Collection)
To remove an element	boolean	remove(Object)
		removeAll(Collection)
		retainAll(Collection)
	void	clear()
To search an element	boolean	contains(Object)
		containsAll(Collection)
To access an element	Iterator	1.iterator() 2.For each loop

miscellaneous		size()
		isEmpty()
		toArray()
		hashCode()
		equals()

Iterator interface

Iterator interface provides the facility of iterating the elements in a forward direction only.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	It returns true if the iterator has more elements otherwise it returns false.
2	public Object next()	It returns the element and moves the cursor pointer to the next element.
3	public void remove()	It removes the last

List Interface

List interface is the child interface of Collection interface. It inhibits a list type data structure in which we can store the ordered collection of objects. It can have duplicate values.

List interface is implemented by the classes ArrayList and LinkedList.

To instantiate the List interface, we must use :

```
1.List <data-type> list1= new ArrayList();
2.List <data-type> list2 = new LinkedList();
```

The classes that implement the List interface are given below.

ArrayList :

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

LinkedList :

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterato itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Queue Interface

Queue interface maintains the first-in-first-out order. It can be defined as an ordered list that is used to hold the elements which are about to be processed. There are various classes like PriorityQueue, Deque, and ArrayDeque which implements the Queue interface.

Queue interface can be instantiated as:

```
1.Queue<String> q1 = new PriorityQueue();
2.Queue<String> q2 = new ArrayDeque();
```

There are various classes that implement the Queue interface, some of them are given below.

PriorityQueue :

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
System.out.println("after removing one element:");
Iterator itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

Set Interface

Set Interface in Java is present in java.util package. It extends the Collection interface. It represents the unordered set of elements which doesn't allow us to store the duplicate items. We can store at most one null value in Set.

Set is implemented by HashSet, LinkedHashSet, and TreeSet.

Set can be instantiated as:

1.Set<data-type> s1 = **new** HashSet<data-type>();
2.Set<data-type> s2 = **new** LinkedHashSet<data-type>();
3.Set<data-type> s3 = **new** TreeSet<data-type>();

HashSet :

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items. Consider the following example.

```
import java.util.*;
public class TestJavaCollection7  {
public static void main(String args[])   {
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Java Non-generic Vs. Generic Collection

Java collection framework was non-generic before JDK 1.5. Since 1.5, it is generic.

Java new generic collection allows you to have only one type of object in a collection. Now it is type-safe, so typecasting is not required at runtime.

Let's see the old non-generic example of creating a Java collection.

→ **ArrayList list=new ArrayList();**//creating old non-generic arraylist

Let's see the new generic example of creating java collection

→ **ArrayList<String> list=new ArrayList<String>();**//creating new generic arraylist

In a generic collection, we specify the type in angular braces. Now ArrayList is forced to have the only specified type of object in it.
If you try to add another type of object, it gives a *compile-time error*.

Difference between ArrayList and LinkedList

ArrayList and LinkedList both implements List interface and maintains insertion order. Both are non synchronized classes. However, there are many differences between ArrayList and LinkedList classes that are given below.

ArrayList	LinkedList
1) ArrayList internally uses a **dynamic array** to store the elements.	LinkedList internally uses a **doubly linked list** to store the elements.
2) Manipulation with ArrayList is **slow** because it internally uses an array. If any element is removed from the array, all the bits are shifted in memory.	Manipulation with LinkedList is **faster** than ArrayList because it uses a doubly linked list, so no bit shifting is required in memory.
3) An ArrayList class can **act as a list** only because it implements List only.	LinkedList class can **act as a list and queue** both because it implements List and Deque interfaces.
4) ArrayList is **better for storing and accessing** data	LinkedList is **better for manipulating** data

HASH SET

- HashSet stores the elements by using a mechanism called **hashing.**
- HashSet contains unique elements only.
- HashSet allows null value.
- HashSet class is non synchronized.
- HashSet doesn't maintain the insertion order. Here, elements are inserted on the basis of their hashcode.
- HashSet is the best approach for search operations.

LINKED HASH SET

- Java LinkedHashSet class contains unique elements only like HashSet.
- Java LinkedHashSet class provides all optional set operation and permits null elements.
- Java LinkedHashSet class is non synchronized.
- Java LinkedHashSet class maintains insertion order.

TREE SET

- Java TreeSet class contains unique elements only like HashSet.
- Java TreeSet class access and retrieval times are quiet fast.
- Java TreeSet class doesn't allow null element.
- Java TreeSet class is non synchronized.
- Java TreeSet class maintains ascending order.

COMPARABLE INTERFACE

COMPARABLE INTERFACE :

Comparable interface is a functional interface(Only one abstract method), since it has only one abstract method.

Method declaration :

public abstract int compareTo(Object o)

STEPS TO MAKE AN OBJECT COMPARABLE TYPE :

STEP 1:

The class must implements java.lang.Comparable interface.

STEP 2:

Override compareTo(Object o) method.

compareTo(Object)

compareTo(Object):

- It is defined in Comparable interface.
- This method is used to compare the current object with passed object.
- The return type is int.

FUNCTION :

- If the value of current object is greater then the passed object it will return positive integer.
- If the value of current object is lesser then the passed object it will return negative integer.
- If the value of current object is equal to the passed object it will return zero.

NOTE :

compareTo() method is called only when the object is of comparable type.

EXAMPLE :

```
class Book implements Comparable
{
        int bid;
        String title;
        double price;
        @Override
        public int compareTo(Object o)
        {
                Book b = (Book)o;
                if(this.price==b.price)
                        return 0;
                else if(this.price>b.price)
                        return 1;
                else
                        return -1;
        }
}
```

- A book object is comparable type.
- The array are collection of book objects can be sorted with the help of built in sort method.
- The built in sort method will sort the objects based on the design provided in compareTo(Object) method.It is known as natural ordering.
- In the above example compareTo(Object) method comparison is done on price of the book.Therefore, the sort method will sort the object based on price only.

DISADVANTAGES OF COMPARABLE

Disadvantage of comparable interface:

- We can sort the object only in one defined order (Natural order)
- If the object is not comparable type, then we can’t sort using built in sort methods.

MAPS

Map is a data structure, which helps the programmers to store the data in the form of key value pairs. Where every value is associated with a unique key.

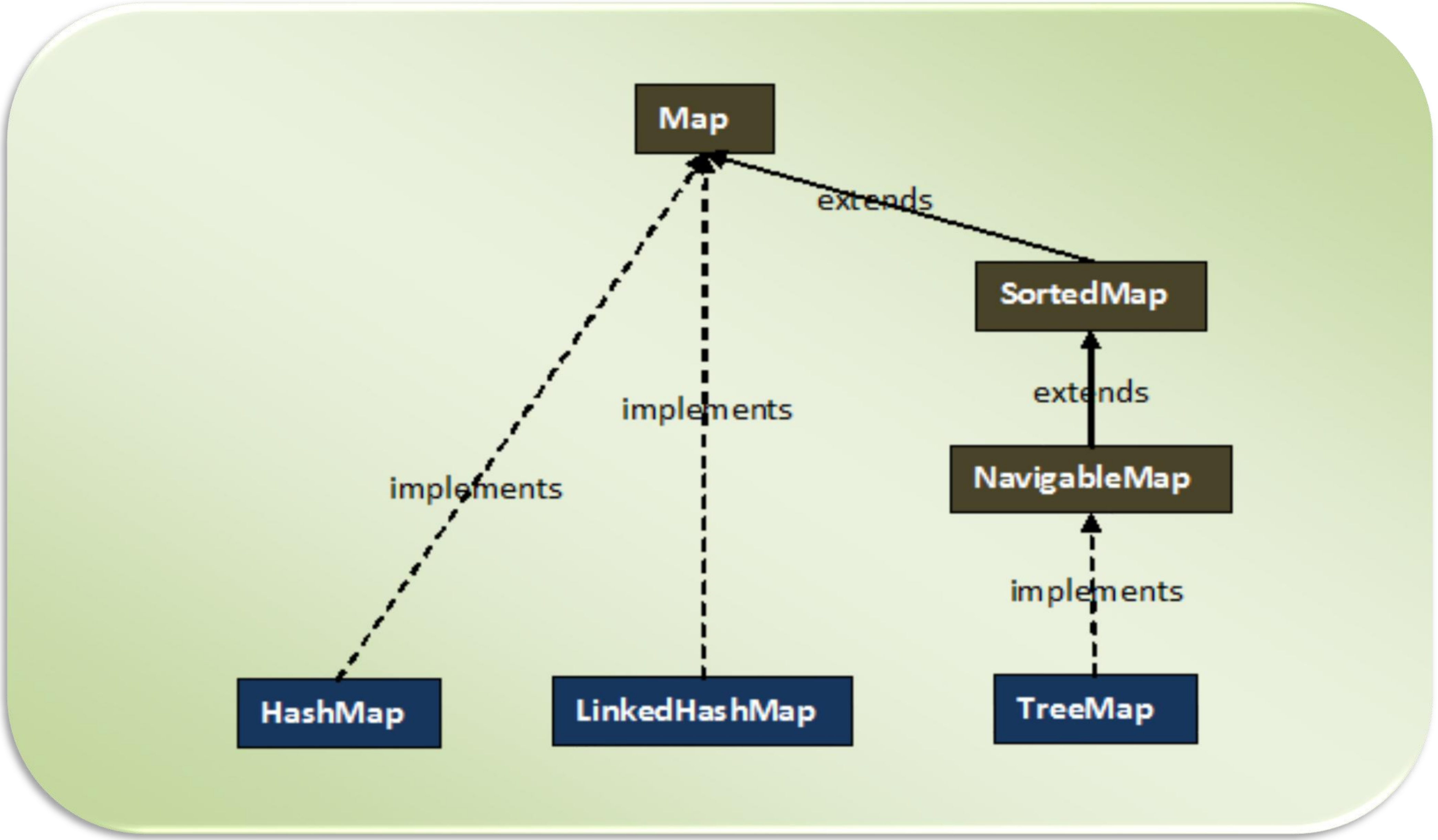
Note :

1. key cannot duplicate.
2. one key can be associated with only one value.

Maps helps us to access the values easily with the help of its' associated key.

Example:

$$\begin{tikzcd}
\text{Key} & \text{value} \\
\text{employee id} \arrow[r] & \text{Employee} \\
\text{pin code} \arrow[r] & \text{City}
\end{tikzcd}$$


Map
extends
SortedMap
implements
extends
implements
NavigableMap
implements
HashMap
LinkedHashMap
TreeMap

1. Map is an interface in java defined in java.util package.
2. We can create generic map by providing the type for both key as well as value, < key_type , value_type >, < K , V >
3. We can obtain 3 different views of a map.
    a. we can obtain a list of values from a map.
    b. We can obtain a set of keys from a map.
    c. We can obtain a set of key-value pairs.

Note : One key value pair is called as an entry or a Mapping.

Methods Of Map interface :

Method Signature	**purpose**
• put(key , value)	1.add an entry to the map (key-value pair) . 2.replace the old value with a new value of an existing entry in the map. putAll(Map) it will copy all the entries from the given map into the current map.
• containsKey(key)	if the key is present returns true else returns false
• containsValue(value)	if the value is present it returns true, else it returns false.
• remove(key)	if the key is present the entry is removed from the map and the value is returned. if the key is not present nothing is removed and null is returned.
• clear()	it removes all the entries in the map.

• get(key)	it is used to access the value associated with a particular key. if the key is present it returns the value. if the key is not present it returns null.
• values()	it returns a collection of values present in the map. return type Collection< v >
• keySet()	it returns a set of keys present in the map return type Set
• entrySet()	it returns a set of all the entries present in the map. return type Set< < k, v> >

- size()
- isEmpty()
- equals()
- hashCode()

Example :

```
public class HashMapDemo {
public static void main(String[] args) {
HashMap<Integer,String> hm=new HashMap();   // generic

// to insert
hm.put(2, "ram");
hm.put(1, "seeta");
hm.put(5, "anil");
hm.put(10, “asha");
hm.put(8, “ravi");

System.out.println(hm);

System.out.println(hm.containsKey(2));

System.out.println(hm.containsValue("seeta"));

hm.remove(1);

System.out.println(hm);

System.out.println(hm.get(5));

System.out.println(hm.values());

System.out.println(hm.keySet());

System.out.println(hm.entrySet());
}
}
```

HashMap :

- It is a concrete implementing class of Map interface.
- data is stored in the form of key=value pair.
- Order of insertion is not maintained.
- key cannot be duplicate, values can be duplicate.
- key can be null.
- value can be null.

TreeMap :

- It is a concrete implementing class of Map interface.
- it also stores data in key=value pair.
- it will sort the entries in the map with respect to keys in ascending order.
- The key in TreeMap must be comparable type. If it is not comparable type we get ClassCastException
- In TreeMap key cannot be null, If it is, null we get NullPointerException.
- In TreeMap key cannot be null, If it is ,null we get NullPointerException.
- A value in TreeMap can be null.

LinkedHashMap

- The **LinkedHashMap Class** is just like HashMap with an additional feature of maintaining an order of elements inserted into it.
- HashMap provided the advantage of quick insertion, search, and deletion but it never maintained the track and order of insertion which the LinkedHashMap provides where the elements can be accessed in their insertion order.

HashTable :

- It is a concrete implementing class of Map interface.
- It is used to store data in key=value format.
- in hashtable the insertion order is not maintained.
- In hashtable both key and value cannot be null. If it is null we get NullPointerException.

ACCESS MODIFIERS

We have two type of modifiers

1. Access modifiers
2. Non access modifiers

Access modifiers :

- Access modifier are responsible to change / modify the accessibility of the member.
- We have four type of access modifiers,

    1. **private**
    2. **default**
    3. **protected**
    4. **Public**

PRIVATE

PRIVATE ACCESS MODIFIER :

- It is a class level modifier , it is applicable for variables , method and constructors.
- If the member of a class is prefixed with private modifier then it is accessible only within the class accessing outside the class is not possible.

EXAMPLE :

```
class A{
private static int i ;
}
class B{
Public static void main(String[] args){
System.out.println(A.i); // CTE
}
```

DEFAULT

DEFAULT ACCESS MODIFIER :

- The accessibility of default modifier is only within the package.It can't be accessed from outside the package.
- If you don't declare any access modifier then it is considered as a default access modifier.

EXAMPLE :

```
package myPack ;

public class Demo{

            static int i ;

}

class Driver{

Public static void main(String[] args){

Demo.i = 5 ; // CTE

}
```

PROTECTED ACCESS MODIFIER :

- The access level of a protected modifier is within the package and outside the package through child class.
- If you do not make the child class, it cannot be accessed from outside the package.

PUBLIC ACCESS MODIFIER :

- The access level of a public modifier is anywhere.
- It can be accessed from within the class, outside the class, within the package as well as outside the package.

SCOPE OF AN ACCESS MODIFIER

ACCESS MODIFIER	WITHIN THE CLASS	WITHIN THE PACKAGE	OUTSIDE THE PACKAGE	OUTSIDE THE PACKAGE BY THE CHILD CLASS
private	YES	NO	NO	NO
default	YES	YES	NO	NO
protected	YES	YES	YES	NO
public	YES	YES	YES	YES

The scope of the access modifier based on the accessibility is :

private < default < protected < public

THREAD

THREAD :

- Thread is lightweight subprocess, smallest unit of a processing which is used to execute the program.
- It uses a shared memory area.
- Every thread will have a components which is required for execution.

Example : Stack, Program Counter, etc,.

JAVA PROCESS CONSIST OF

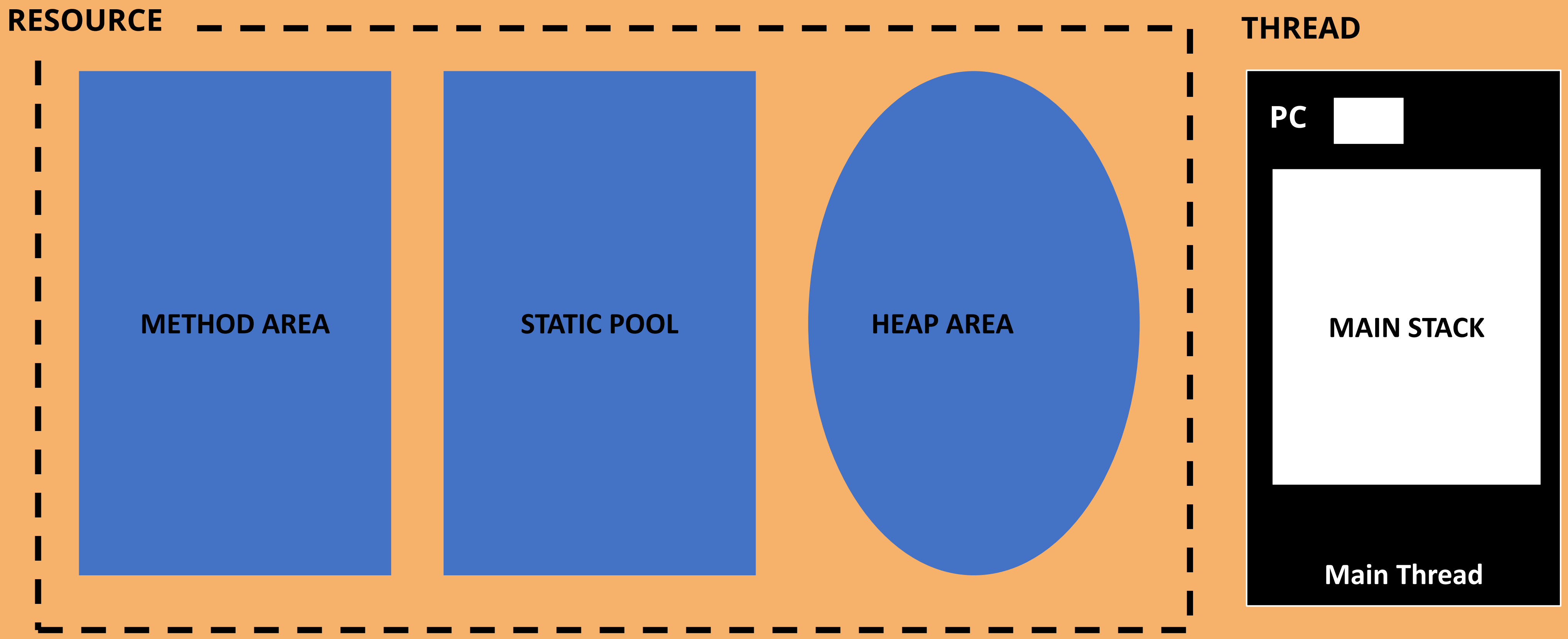
Java process consist of :

- Every process will have default thread for execution, known as Main thread.
- Every process will have a memory to store the resource (Code segment, data segment).

    Code segment - instructions to be stored.

    Data segment - datas are stored.

JAVA RUNTIME EXECUTION PROCESS :



- Every threads will use the common resource.
- main(String[] args) method is always executed by main thread.
- PC-Contains a reference of the next instruction.

MAIN THREAD

MAIN THREAD :

The execution of a main thread always starts from main(String[] args) method and ends at main(String[] args) Method**.**

Attributes of a thread :

- Name
- Thread ID
- Priority, etc,.

THREAD CLASS

THREAD CLASS :

- In java, we have a class that defines a thread that class is known as Thread class present in java.lang package.
- In thread class name, id, priority, state, etc,. of a thread is defined.
- Thead class has a built in methods to perform actions on thread.
- Thread class is an encapsulated class, all the attributes are made private, we can access them through getter and setter methods.

DECLARATION OF THREAD CLASS:

Thread class declared in java.lang package.

public class Thread extends Object implements Runnable

STATIC METHODS

Return type	Method signature	Function
Thread	currentThread()	Returns the reference of the currently executing thread
boolean	interrupted()	Tests whether the current thread has been interrupted.
void	sleep(long millis)	Cause the currently executing thread to sleep for a specified time.
void	sleep(long millis, int nanos)	Cause the currently executing thread to sleep for a specified time.
void	yield()	A hint to the processor that current thread is willing to yield its current use of a processor.

NON STATIC METHODS

Return type	Method signature	Function
String	getName()	Returns this thread name
long	getId()	Returns this identifier of a thread
int	getPriority()	Returns this thread’s priority
ThreadGroup	getThreadGroup()	Returns thread group to which this thread belongs to
Thread.state	getState()	Returns the state of this thread
void	interrupt()	Interrupts this thread
boolean	isAlive()	Tests if this thread is alive
boolean	isDaemon()	Tests if this thread is daemon
boolean	isInterrupted()	Tests whether this thread is interrupted

NON STATIC METHODS

Return type	Method signature	Function
void	join()	Waits for this thread to die
void	join(long millis)	Waits at most millis millisecond for this thread to die
void	join(long millis, int nanos)	Waits at most millis millisecond plus nanosecond for this thread to die
void	resume()	This method exist for solely for use with suspend()
void	run()	The execution of thread we created is starts from run() method and eds at run().
void	setDaemon(boolean on)	Make this thread as a daemon
void	setPriority(int newPriority)	Changes the priority of this thread
void	start()	Once the start method is called the thread is ready to execute
void	stop()	
void	suspend()	

To access the properties of Main Thread:

- Thread.currentThread()
- getName()
- getPriority()

EXAMPLE:

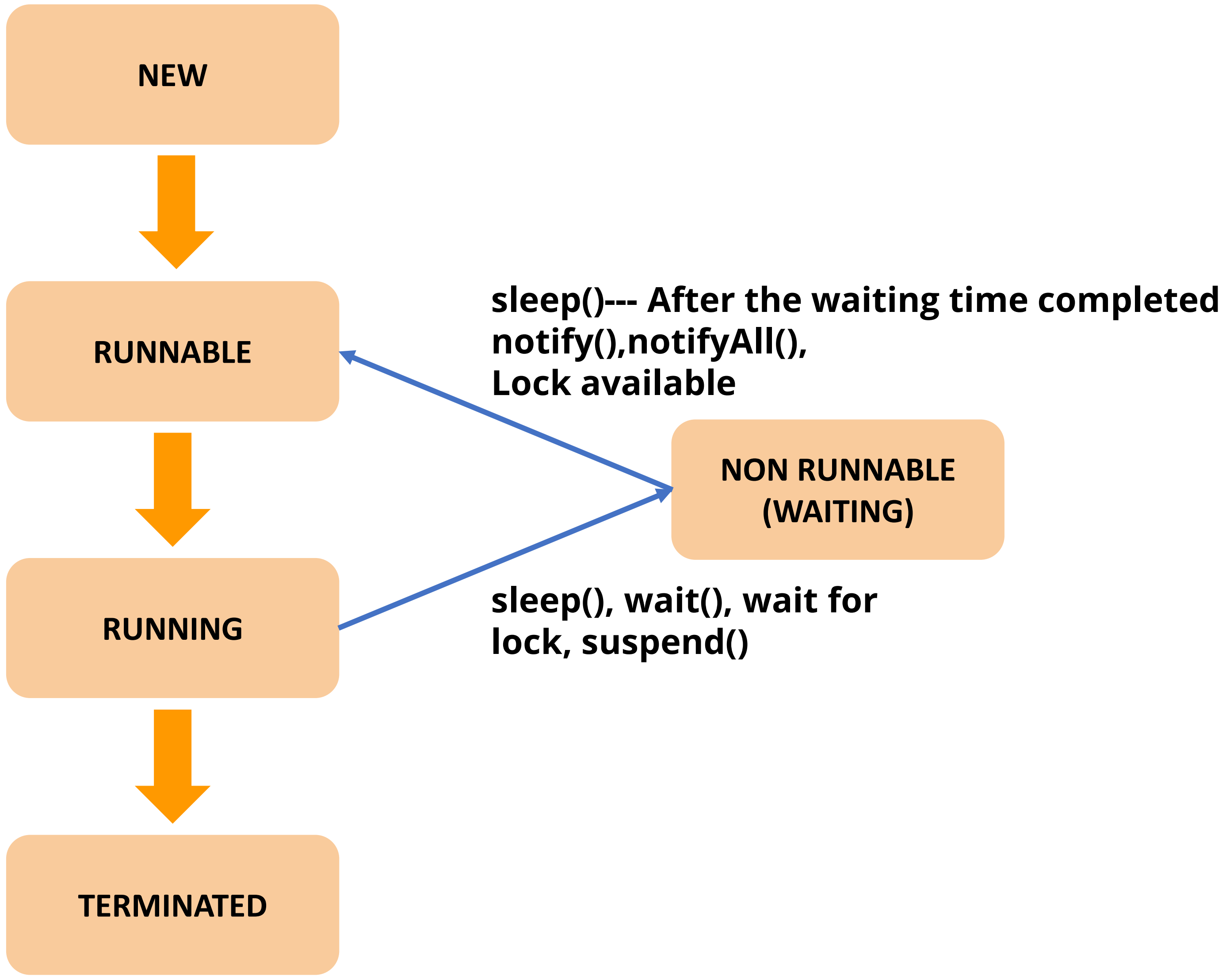
```
class ThreadPropertyDemo
{
        public static void main(String[] args)
        {
                System.out.println(Thread.currentThread().getName());
//main
                System.out.println(Thread.currentThread().getId());

        }
}
```

LIFE CYCLE OF A THREAD

Stages of threads :

- Born **(New)**
- Ready **(Runnable)**
- Executing **(Running)**
- Waiting **(Non runnable)**
- Dead **(Terminated)**


NEW
RUNNABLE
RUNNING
TERMINATED
NON RUNNABLE (WAITING)
sleep()--- After the waiting time completed
notify(),notifyAll(),
Lock available
sleep(), wait(), wait for
lock, suspend()

NEW:

Thread object is created.

RUNNABLE:

By calling start() method thread goes to runnable stage.

RUNNING:

When Thread Scheduler selects the thread for execution we say thread is under execution.

TERMINATED:

- Once execution of the of a thread is successfully completed it goes to dead stage.
- A thread can also be stopped when forcefully send to dead stage.

WAIT STAGE:

- A thread which is under execution if interrupted with the help of methods such as sleep(), wait(, etc,.
- A thread can also goes to wait stage if the required resource is not available.

Java has a beautiful mechanism to create a multiple threads with the help of Thread class.

Why programmer should create a multiple thread ?

- To achieve parallelism
- To reduce a execution time
- Efficient use of resource

BY EXTENDING THREAD CLASS

Creating a thread by extending Thread class :

- Create a class by extending java.lang.Thread class
- Override the run() Method

Executing a thread:

- Inside a main method of any class create an object for the class that extends Thread and overrides runnable.
- Call the start method by using object reference of that instantiated class.

EXAMPLE :

```
class MyThread extends Thread
{
        @Override
        public void run()
        {
                System.out.println(currentThread().getName()+" is executing run()");
                System.out.println("Priority is "+currentThread().getPriority());
        }
        public static void main(String[] args)
        {
                MyThread t = new MyThread();
                t.start();
        }
}
```