UNIT – 1 Object Oriented Programming

PRINCIPLES OF OOP:

- **Class and Objects, Encapsulation, Data Abstraction, Polymorphism and Inheritance.**

OBJECT:

An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Object is an instance of a class. Class is a template or blueprint from which objects are created. So object is the instance(result) of a class.

CLASS:

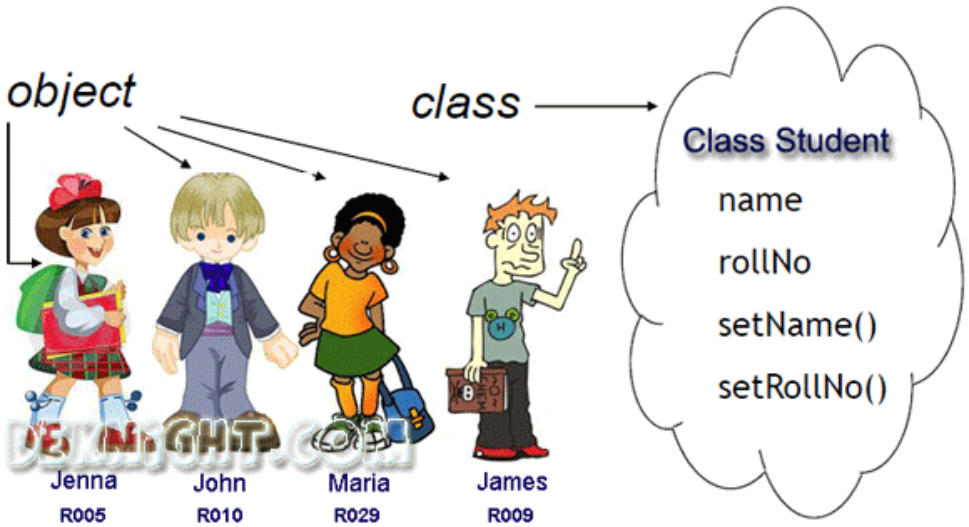
A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.

A class in java can contain:

- data member
- method
- constructor
- block
- class and interface

Syntax to declare a class:

```
class <class_name>
{
   data member;
   method;
}
```



ENCAPSULATION:

Encapsulation means that the internal representation of an object is generally hidden from view outside of the object's definition.

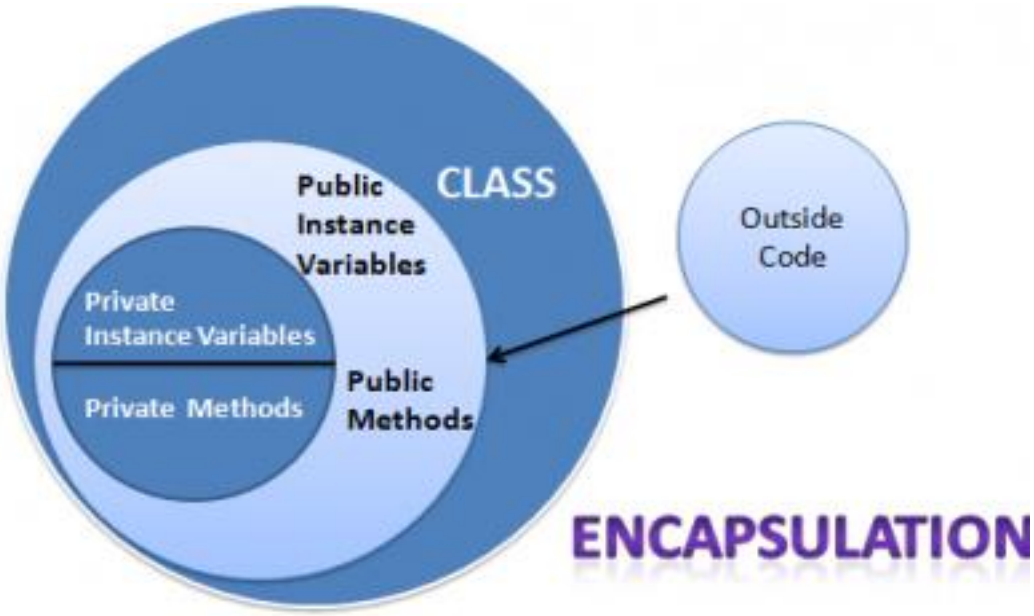
Typically, only the object's own methods can directly inspect or manipulate its fields. Encapsulation is the hiding of data implementation by restricting access to accessors and mutators.

An accessor is a method that is used to ask an object about itself. In OOP, these are usually in the form of properties, which have a *get* method, which is an accessor method. However, accessor method share not restricted to properties and can be any public method that gives information about the state of the object.

A Mutator is public method that is used to modify the state of an object, while hiding the implementation of exactly how the data gets modified. It's the *set* method that lets the caller modify the member data behind the scenes.

Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. This type of data protection and implementation protection is called *Encapsulation*.

A benefit of encapsulation is that it can reduce system complexity.

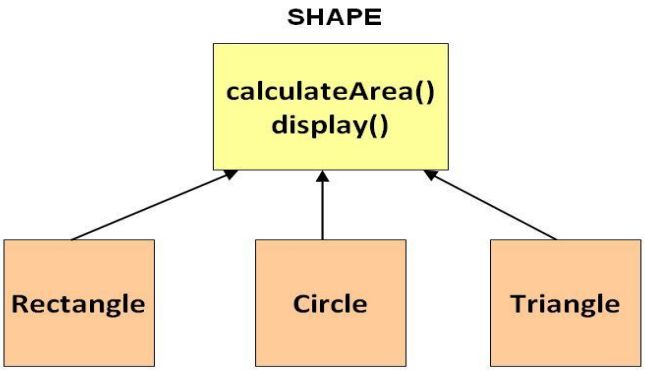


ABSTRACTION

Data abstraction and encapsulation are closely tied together, because a simple definition of data abstraction is the development of classes, objects, types in terms of their interfaces and functionality, instead of their implementation details.

Abstraction denotes a model, a view, or some other focused representation for an actual item. "An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer".

In short, data abstraction is nothing more than the implementation of an object that contains the same essential properties and actions we can find in the original object we are representing.



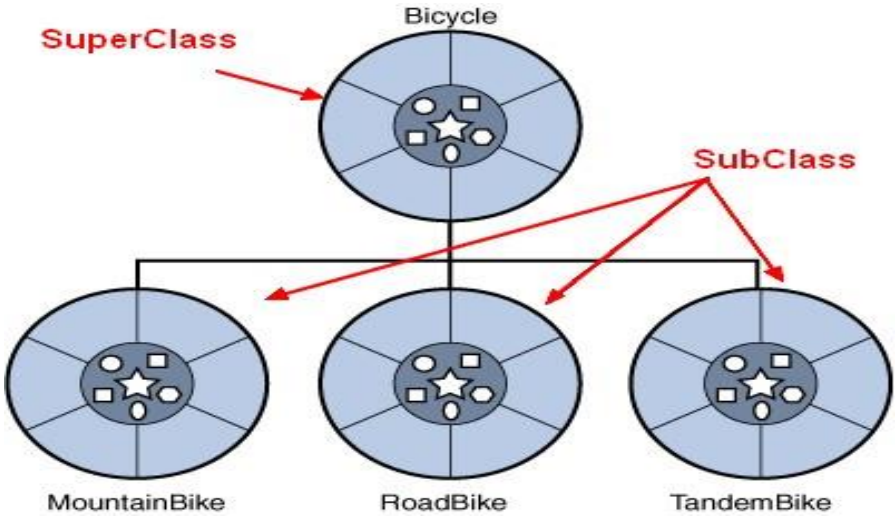
INHERITANCE

Inheritance is a way to reuse code of existing objects, or to establish a subtype from an existing object, or both, depending upon programming language support. In classical inheritance here objects are defined by classes, classes can inherit attributes and behavior from pre-existing classes called base classes, superclasses, parent classes or ancestor classes. The resulting classes are known as derived classes, subclasses or child classes. The relationships of classes through inheritance gives rise to a hierarchy.

Subclasses and Super classes. A subclass is a modular, derivative class that inherits one or more properties from another class (called the superclass).

The properties commonly include class data variables, properties, and methods or functions. The superclass establishes a common interface and foundational functionality, which specialized subclasses can inherit, modify, and supplement.

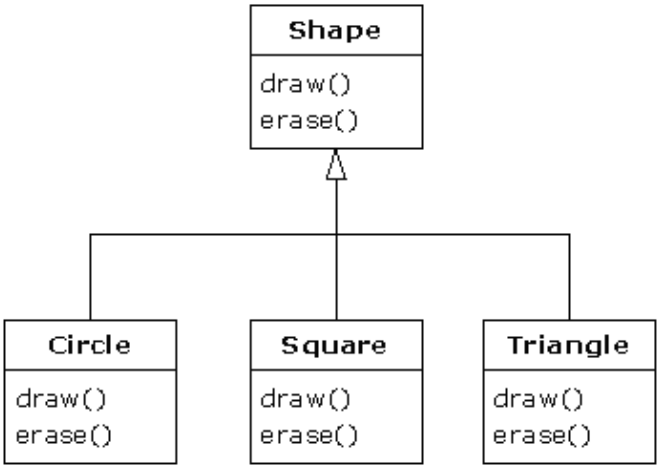
The software inherited by a subclass is considered reused in the subclass. In some cases, a subclass may customize or redefine a method inherited from the superclass. A superclass method which can be redefined in this way is called a virtual method.



POLYMORPHISM

Polymorphism means one name, many forms. Polymorphism manifests itself by having multiple methods all with the same name, but slightly different functionality. There are 2 basic types of polymorphism. Overriding, also called run-time polymorphism.

For method overloading, the compiler determines which method will be executed, and this decision is made when the code gets compiled. Overloading, which is referred to as compile-time polymorphism. Method will be used for method overriding is determined at runtime based on the dynamic type of an object.



PROGRAM STRUCTURE IN JAVA:

Introduction:

Java is an object-oriented programming, platform-independent, and secure programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications.

A typical structure of a Java program contains the following elements:

- Documentation Section
- Package Declaration
- Import Statements
- Interface Section
- Class Definition
- Class Variables and Variables
- Main Method Class
- Methods and Behaviours

Documentation Section:

The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program.

- **Single-line Comment:** It starts with a pair of forwarding slash **(//)**.

For example: //My First Java Program

Multi-line Comment: It starts with a **/*** and ends with ***/.** We write between these two symbols. **For example:**

```
/*It is an example of
    multiline comment*/
```

Documentation Comment: It starts with the delimiter **(/**)** and ends with ***/**.

For example: /**It is an example of documentation comment*/

Package Declaration:

The package declaration is optional. It is placed just after the documentation section. In this section, we declare the **package name** in which the class is placed. Note that there can be **only one package** declaration statement in a Java program.

For Example:

package javatpoint; //where javatpoint is the package name
package com.javatpoint; //where com is the root directory & javatpoint is the subdirectory

Import Statements

The package contains many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. We use the **import** keyword to import the class.

For Example:

import java.util.Scanner; //it imports the Scanner class only
import java.util.*; //it imports all the class of the java.util package

Interface Section

It is an optional section. We can create an **interface** in this section if required. We use the **interface** keyword to create an interface. An interface is a slightly different from the class. It contains only **constants** and **method** declarations.

For Example:

```
interface car
{
    void start();
    void stop();
}
```

Class Definition

In this section, we define the class. It is a **vital** part of a Java program. Without the class, we cannot create any Java program. A Java program may contain more than one class definition. We use the **class** keyword to define the class.

For Example:

```
class Student //class definition
{
}
```

WRITING SIMPLE JAVA PROGRAM:

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

Explanation:

1. **public class HelloWorld:** This line declares a public class named HelloWorld. This is the entry point of the Java program.
2. **public static void main(String[] args):** This is the main method, which is the starting point for execution in a Java program. It takes an array of strings as arguments (args) but is not used in this simple example.
3. **System.out.println("Hello, World!");:** This line prints the message "Hello, World!" to the console.

To run this program:

1. Save the code in a file named HelloWorld.java.
2. Open a terminal or command prompt and navigate to the directory where you saved the file.
3. Compile the code using the javac compiler:
   javac HelloWorld.java
4. Run the compiled class using the java command:
   java HelloWorld
   Output: Hello, World!

JAVA TOKENS

In Java, a token is the smallest program unit with a meaning. They are the fundamental building blocks of Java code. Here are the different types of tokens in Java:

1. Identifiers:

- Names given to variables, methods, classes, and interfaces.
- Must start with a letter or underscore (_) and can contain letters, digits, underscores, and dollar signs ($).
- Cannot be a keyword.

2. Keywords:

- Reserved words in Java that have specific meanings.
- Examples: if, else, for, while, class, public, private, static, int, double, String, etc.

3. Literals:

- Direct representations of values in the source code.
- Can be of different types:
  - Integer literals: 123, 0x45, 0b101
  - Floating-point literals: 3.14, 1.2e3
  - Character literals: 'a', '\n' (newline)
  - String literals: "Hello, world!"
  - Boolean literals: true, false
  - Null literal: null

4. Operators:

- Symbols used to perform operations on values.
- Examples: +, -, *, /, %, ==, !=, <, >, <=, >=, &&, ||, !, =, +=, -=, *=, /=, %=

5. Separators:

- Characters used to separate different elements of the program.
- Examples: {}, (), [], ;, ,, .

6. Comments:

- Text that is ignored by the compiler.
- Used to explain the code.
- Single-line comments: // This is a comment
- Multi-line comments: /* This is a multi-line comment */

Example:

```
public class MyClass
{
   public static void main(String[] args)
   {
      int x = 10; // Identifier, keyword, literal, operator, separator
      String message = "Hello, world!"; // Identifier, keyword, literal, separator
      if (x > 5)
      {
         // Keywords, operator, separator
         System.out.println(message); // Identifier, keyword, separator, literal
      }
    }
}
```

JAVA STATEMENTS

In Java, a statement is a unit of code that performs an action or expresses a declaration. Here are the different types of statements in Java:

1. Declaration Statements:

- Used to declare variables, methods, classes, and interfaces.

Examples:

```
int x = 10; // Variable declaration
public class MyClass
{ // Class declaration
   public void myMethod()
   { // Method declaration
      // ...
   }
}
```

2. Expression Statements:

- Used to evaluate expressions and perform actions based on the result.
- Examples:

```
x = 20; // Assignment expression
System.out.println("Hello, world!"); // Method invocation expression
```

3. Control Flow Statements:

- Used to control the flow of execution in a program.
- Examples:
    - **Conditional statements:**

```
if (condition) {
   // Statements to execute if condition is true
} else {
   // Statements to execute if condition is false
}
```

- **Loop statements:**

```
for (initialization; condition; update) {
   // Statements to execute
}
while (condition) {
   // Statements to execute
}
do {
   // Statements to execute
} while (condition);
```

- **Jump statements:**

```
break; // Breaks out of a loop or switch statement
continue; // Skips the current iteration of a loop
return; // Returns from a method
```

4. Block Statements:

- Used to group multiple statements together within curly braces ({}).

Examples:

```
if (condition) {
   // Block of statements
}
```

5. Try-Catch-Finally Statements:

- Used for exception handling.

Examples:

```
try {
   // Code that might throw an exception
} catch (Exception e) {
   // Handle the exception
} finally {
   // Code to execute regardless of whether an exception is thrown
}
```

Example:

```
public class MyProgram {
   public static void main(String[] args) {
      int x = 10; // Declaration statement
```

```
        if (x > 5) { // Conditional statement
            System.out.println("x is greater than 5"); // Expression statement
        } else {
            System.out.println("x is less than or equal to 5");
        }
    }
}
```

COMMAND-LINE ARGUMENTS:

- **Input to Java Programs:** Command-line arguments are values passed to a Java program when it is executed from the command line. They provide a way for users to interact with the program and provide input data.
- **Accessing Arguments:** The main method of a Java class receives an array of strings (String[] args) as a parameter. This array contains the command-line arguments passed to the program.
- **Indexing Arguments:** You can access individual command-line arguments using their index in the args array. The first argument has an index of 0, the second has an index of 1, and so on.

Example:

```
public class CommandLineArguments {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println("Number of arguments:" + args.length);
            for (int i = 0; i < args.length; i++) {
                System.out.println("Argument " + (i + 1) + ": " + args[i]);
            }
        } else {
            System.out.println("No arguments provided.");
        }
    }
}
```

Example Usage:

java CommandLineArguments arg1 arg2 arg3

In this example, arg1, arg2, and arg3 are the command-line arguments passed to the CommandLineArguments program. The program can then access and process these arguments using the args array.

USER INPUT:

- **Obtaining Data from Users:** User input allows programs to interact with the user and obtain data dynamically.
- **Scanner Class:** The Scanner class is commonly used to read user input from the console.

Reading User Input:

1. **Create a Scanner Object:**

```
Scanner scanner = new Scanner(System.in);
```

2. **Read Input:**
    - **Integers:**
      ```
      int number = scanner.nextInt();
      ```
    - **Double-precision numbers:**
      ```
      double decimal = scanner.nextDouble();
      ```
    - **Strings:**
      ```
      String text = scanner.nextLine();
      ```

Example:

```
import java.util.Scanner;
public class UserInputExample
{
	public static void main(String[] args)
	{
		Scanner scanner = new Scanner(System.in);
		System.out.print("Enter your name: ");
		String name = scanner.nextLine();
		System.out.print("Enter your age: ");
		int age = scanner.nextInt();
		System.out.println("Hello, " + name + "! You are " + age + " years old.");
	}
}
```

ESCAPE SEQUENCES:

- **Special Characters:** Escape sequences are special character combinations used to represent characters that are difficult or impossible to represent directly in a Java string.
- **Common Escape Sequences:**
    - \n: Newline
    - \t: Tab
    - \r: Carriage return
    - \b: Backspace
    - \f: Form feed
    - \\: Backslash
    - \': Single quote
    - \": Double quote
- **Unicode Escape Sequences:**
    - \uXXXX: Represents a Unicode character, where XXXX is a four-digit hexadecimal code point.

Comments:

- **Ignored by Compiler:** Comments are text that is ignored by the Java compiler. They are used to explain the code and make it more readable.
- **Types of Comments:**
    - **Single-line comments:** Start with // and continue until the end of the line.

- **Multi-line comments:** Start with /* and end with */.
- **Javadoc comments:** Start with /** and end with */. They are used to generate documentation for Java classes, methods, and fields.

Example:

```
// This is a single-line comment

/* This is a
multi-line comment */

/**
 * This is a Javadoc comment.
 * @param name The name of the person
 */
```

PROGRAMMING STYLE:

- **Consistency and Readability:** Programming style refers to the conventions and guidelines followed to write clean, readable, and maintainable code. It involves consistent formatting, naming conventions, and coding practices.
- **Code Formatting:**
  - Indentation: Use consistent indentation (e.g., 4 spaces) to improve code readability.
  - Line Length: Limit line length to a reasonable value (e.g., 120 characters) to avoid horizontal scrolling.
  - Spacing: Use consistent spacing around operators, keywords, and parentheses.
- **Naming Conventions:**
  - Class Names: Start with an uppercase letter and use PascalCase for subsequent words (e.g., MyClassName).
  - Variable and Method Names: Start with a lowercase letter and use camelCase for subsequent words (e.g., myVariable, myMethod).
  - Constants: Use all uppercase letters with underscores between words (e.g., MAX_VALUE).
- **Comments:**
  - Explain Purpose: Use comments to explain the purpose of code sections that are not self-explanatory.
  - Avoid Redundancy: Don't comment on obvious code.
  - Update Comments: Keep comments up-to-date as the code changes.
- **Code Organization:**
  - Modularization: Break down code into smaller, reusable modules (classes, methods).
  - Naming Consistency: Use consistent naming conventions for related variables, methods, and classes.
- **Error Handling:**
  - Exception Handling: Use try-catch blocks to handle potential exceptions and prevent program crashes.
- **Code Review:**

- Peer Review: Have your code reviewed by others to identify potential issues and improve quality.

DATA TYPES IN JAVA

Data types in Java define the kind of values a variable can hold and the operations that can be performed on them. Java has two main categories of data types: primitive data types and reference data types.

Primitive Data Types:

- **Numeric Types:**
  - byte: 8-bit signed integer (-128 to 127)
  - short: 16-bit signed integer (-32,768 to 32,767)
  - int: 32-bit signed integer (-2,147,483,648 to 2,147,483,647)
  - long: 64-bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
  - float: 32-bit floating-point number (approximately ±3.4E-38 to ±3.4E+38)
  - double: 64-bit floating-point number (approximately ±1.7E-308 to ±1.7E+308)
- **Character Type:**
  - char: 16-bit Unicode character (0 to 65,535)
- **Boolean Type:**
  - boolean: Represents true or false values

Reference Data Types:

- **Classes:**
  - Blueprints for objects.
  - Examples: String, Integer, Double, Character, Boolean, Array, etc.
- **Interfaces:**
  - Contracts that define the behavior of classes.
- **Arrays:**
  - Ordered collections of elements of the same type.

VARIABLE DECLARATION IN JAVA

In Java, variables are used to store data values. To declare a variable, you specify its data type and a unique name. The data type determines the kind of values the variable can hold, while the name is used to reference the variable.

Syntax:

```
dataType variableName;
```

Example:

```
int age;
String name;
double salary;
```

LITERAL CONSTANTS IN JAVA

Literal constants are values that are directly specified in the source code. They represent specific data values that are not stored in variables.

Types of Literal Constants:

- **Integer literals:** Represent integer values.

    - Decimal form: 123, -456
    - Octal form (starts with 0): 0123 (equivalent to decimal 83)
    - Hexadecimal form (starts with 0x): 0xABC (equivalent to decimal 2748)
    - Binary form (starts with 0b): 0b1011 (equivalent to decimal 11)
- **Floating-point literals:** Represent decimal numbers.
    - Decimal form: 3.14, -2.5
    - Scientific notation: 1.2e3 (equivalent to 1200), 3.4E-5 (equivalent to 0.000034)
- **Character literals:** Represent single characters enclosed in single quotes (').
    - 'A', 'a', '#', '\n' (newline)
- **String literals:** Represent sequences of characters enclosed in double quotes (").
    - "Hello, world!", "This is a string"
- **Boolean literals:** Represent boolean values.
    - true, false
- **Null literal:** Represents the absence of a value.
    - null

SYMBOLIC CONSTANTS

Note: Symbolic constant is same as Final Variable.

PRECEDENCE AND ASSOCIATIVITY OF OPERATORS IN JAVA

Operator precedence and associativity determine the order in which operations are performed in a Java expression.

Precedence:

- The order in which operators are evaluated.
- Higher precedence operators are evaluated first.

Associativity:

- The order in which operators of the same precedence are evaluated.
- Left-associative operators are evaluated from left to right.
- Right-associative operators are evaluated from right to left.

Operator Precedence Table:

Operator	Associativity
() (parentheses)	Right-associative
++, -- (postfix)	Left-associative
++, -- (prefix)	Right-associative
!, ~ (unary)	Right-associative
*, /, % (multiplication, division, modulus)	Left-associative
+, - (addition, subtraction)	Left-associative
<, >, <=, >= (comparison)	Left-associative
==, != (equality)	Left-associative
& (bitwise AND)	Left-associative
^ (bitwise XOR)	Left-associative
`	` (bitwise OR)

&& (logical AND)	Left-associative
`	` (logical OR)
? : (ternary operator)	Right-associative
= (assignment)	Right-associative
+=, -=, *=, /= (compound assignment)	Right-associative

Example:

int result = 2 * 3 + 5; // Equivalent to (2 * 3) + 5

In this example, the multiplication (*) operator has higher precedence than the addition (+) operator, so 2 * 3 is evaluated first. The result of the multiplication (6) is then added to 5, resulting in a final value of 11.

Bitwise Operators

Bitwise operators are used to perform the manipulation of individual bits of a number. They can be used with any integral type (char, short, int, etc.).

Bitwise AND (&)

This operator is a binary operator, denoted by '&.' It returns bit by bit AND of input values, i.e., if both bits are 1, it gives 1, else it shows 0.

Example:

a = 5 = 0101 (In Binary)
b = 7 = 0111 (In Binary)

Bitwise AND Operation of 5 and 7

```
  0101
& 0111
_______
0101 = 5 (In decimal)
```

Bitwise OR (|)

This operator is a binary operator, denoted by '|'. It returns bit by bit OR of input values, i.e., if either of the bits is 1, it gives 1, else it shows 0.

Example:

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise OR Operation of 5 and 7

```
  0101
| 0111
_______
```

0111 = 7 (In decimal)

Bitwise XOR (^)

This operator is a binary operator, denoted by '^.' It returns bit-by-bit XOR of input values, i.e., if corresponding bits are different, it gives 1, else it shows 0.

Example:

a = 5 = 0101 (In Binary)

b = 7 = 0111 (In Binary)

Bitwise XOR Operation of 5 and 7

```
  0101
^ 0111
--------
  0010 = 2 (In decimal)
```

Bitwise Complement (~)

This operator is a unary operator, denoted by '~.' It returns the one's complement representation of the input value, i.e., with all bits inverted, which means it makes every 0 to 1, and every 1 to 0.

Example:

a = 5 = 0101 (In Binary)

Bitwise Complement Operation of 5

~ 0101

1010 = 10 (In decimal)

Bit-Shift Operators (Shift Operators)

Shift operators are used to shift the bits of a number left or right, thereby multiplying or dividing the number by two, respectively. They can be used when we have to multiply or divide a number by two.

Syntax:

number **shift_op** number_of_places_to_shift;

Types of Shift Operators:

Shift Operators are further divided into 2 types. These are:

1. Right shift operator (>>)

- The right shift operator shifts all the bits to the right. The empty space in the left side is filled depending on the input number:

2. Left shift operator(<<)

- The left shift operator shifts the bits to the left by the number of times specified by the right side of the operand. After the left shift, the empty space in the right is filled with 0.

```
import java.util.Scanner;
public class BitwiseOperators
{
	public static void main(String[] args)
	{
		Scanner input = new Scanner(System.in);
		System.out.print("Enter first number: ");
		int num1 = input.nextInt();
		System.out.print("Enter second number: ");
		int num2 = input.nextInt();
		System.out.println("Bitwise AND: " + (num1 & num2));
		System.out.println("Bitwise OR: " + (num1 | num2));
		System.out.println("Bitwise XOR: " + (num1 ^ num2));
		System.out.println("Bitwise NOT: " + (~num1));
		System.out.println("Bitwise Left Shift: " + (num1 << 2));
		System.out.println("Bitwise Right Shift: " + (num1 >> 2));
	}
}
```

Output:

```
Enter first number: 5
Enter second number: 7
Bitwise AND: 5
Bitwise OR: 7
Bitwise XOR: 2
Bitwise NOT: -6
Bitwise Left Shift: 20
Bitwise Right Shift: 1
```

Data Types in Java:

Java is a strongly typed language, meaning every variable must have a specific data type declared. The data type determines the size of the variable and the range of values it can hold.

Primitive Data Types:

Java has eight primitive data types, which are the most basic building blocks for creating variables. They are divided into two categories:

Numerical Data Types:

- **Integer Types:**
  - byte: 8-bit signed integer (-128 to 127)
  - short: 16-bit signed integer (-32,768 to 32,767)
  - int: 32-bit signed integer (-2,147,483,648 to 2,147,483,647)
  - long: 64-bit signed integer (-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)

- **Floating-Point Types:**
  - float: 32-bit floating-point number (approximately 1.4E-45 to 3.4E+38)
  - double: 64-bit floating-point number (approximately 4.9E-324 to 1.7E+308)

Character Data Type:

- char: 16-bit Unicode character (0 to 65,535)

Boolean Data Type:

- boolean: Represents **true** or **false** values.

Non-Primitive Data Type:

•**String:** Represents a sequence of characters. It is not a primitive type but a class in Java.

VARIABLE DECLARATION IN JAVA

In Java, the following syntax is used to declare variables:

- dataType variableName;

where:

- dataType specifies the type of data the variable can hold (e.g., int, double, String, boolean).
- variableName is the identifier given to the variable.

For Example:

- int x;

TYPE CASTING IN JAVA:

- Type casting is the process of converting a variable from one data type to another. It's essential in Java to ensure that data is used correctly and efficiently.

Implicit Casting:
• Occurs automatically when a value of a smaller data type is assigned to a variable of a larger data type.
• This is generally safe because there's no loss of precision.

Example:
Java
```
int x = 10;
double y = x; // Implicit casting from int to double
```

Explicit Casting (Narrowing):
• Requires a manual conversion using the cast operator ((dataType)).
• Can potentially lead to loss of precision or data truncation if the value cannot be accurately represented in the smaller data type.

Example:
Java
```
double x = 10.5;
int y = (int) x;
```

```
public class DataTypesExample
{
    public static void main(String[] args)
   {
        int i=5;
        byte b=10;
        float f=10.5f;
        System.out.println("byte: " + b);
        System.out.println("int: " + i);
        System.out.println("float: " + f);
        i=b; // implicit conversion
        b=(byte)i; // explicit conversion
        i=f; // Loss of Precision
        // Printing the values after casting
        System.out.println("byte: " + b);
        System.out.println("int: " + i);
        System.out.println("float: " + f);
    }
}
```

Output: error

```
        i=f; // Loss of Precision
```

```
public class DataTypesExample
{
    public static void main(String[] args)
   {
        int i=5;
        byte b=10;
        float f=10.5f;
        System.out.println("byte: " + b);
        System.out.println("int: " + i);
        System.out.println("float: " + f);
        i=b; // implicit conversion
        b=(byte)i; // explicit conversion
        i=(int)f;
        // Printing the values after casting
        System.out.println("byte: " + b);
        System.out.println("int: " + i);
        System.out.println("float: " + f);
    }
}
```

Output:

```
byte: 10
int: 5
float: 10.5
byte: 10
int: 10
float: 10.5
```

SCOPE OF VARIABLES IN JAVA

The scope of a variable is the part of the program where the variable is accessible. Each variable has a scope that specifies how long it will be seen and used in a programme.

There are three scopes for variables in java.

1. **Method level scope** (Variable declared within the method)
2. **Block level scope** (Variable declared within the block)
3. **Class level scope** (Variable declared within the Class)

Method level scope:

- Variables declared inside a method have method-level scope and can't be accessed outside the method.

Example:

```
void display()
{
        int i=10; // This scope of this variable is within this method only
}
```

Block Level Scope:

- A variable declared inside pair of brackets “{” and “}” in a method has scope within the brackets only.

Example:

```
for(int i=0;i<=10;i++)
{
        //This variable scope is within the loop or brackets { }
}
```

Class Level Scope:

- These variables must be declared inside a class (ie. Outside any function). They can be directly accessed anywhere in the class.

Example:

```
class ClassScope
{
        int x; //This variable scope is anywhere inside the class.
}
```

```
class ScopeInJava
{		int x=10; // class scope variable
		void display()
		 {		int i; // method scope variable
				i=10;
				for(int j=1;j<=i;j++)
				{		// block scope variable
						System.out.println("the value of j = "+j);
				}
				System.out.println("I value is = "+i);
				System.out.println("x value is = "+x);
				System.out.println("j value is = "+j);
		}
		public static void main(String[] args)
		{		MethodScope ms=new MethodScope();
				ms.display();
				System.out.println("I value is = "+i);
				System.out.println("x value is = "+x);
				System.out.println("j value is = "+j);
		}
}
```

LITERAL:

- Any constant value which can be assigned to the variable is called literal / constant.

Types of Literals in Java:

1. Integral Literals:

- Integral literals consist of digit sequences and are broken down into these sub-types:

•**Decimal Integer:** Decimal integers use a base ten and digits ranging from 0 to 9. They can have a negative (-) or a positive (+), but non-digit characters or commas aren't allowed between characters. Example: 2022, +42, -68.

•**Octal Integer:** Octal integers use a base eight and digits ranging from 0 to 7. Octal integers always begin with a "0." Example: 007, 0295.

•**Hexa-Decimal:** Hexa-decimal integers work with a base 16 and use digits from 0 to 9 and the characters of A through F. The characters are case-sensitive and represent a 10 to 15 numerical range. Example: 0xf, 0xe.

•**Binary Integer:** Binary integers uses a base two, consisting of the digits "0" and "1." The prefix "0b" represents the Binary system. Example: 0b11011.

2. Floating-Point Literals:

Floating-point literals are expressed as exponential notations or as decimal fractions. They can represent either a positive or negative value, but if it's not specified, the value defaults to positive. Floating-point literals come in these formats.

Floating:
Floating format single precision (4 bytes) end with an "f" or "F."

- Example: 4f.

Floating format double precision (8 bytes) end with a "d" or "D."

- Example: 3.14d.

3. Char Literals:

Character (Char) literals are expressed as an escape sequence or a character, enclosed in single quote marks, and always a type of character in Java. Char literals are sixteen-bit Unicode characters ranging from 0 to 65535.

- Example: char ch = 077.

4. String Literals:

String literals are sequences of characters enclosed between double quote ("") marks. These characters can be alphanumeric, special characters, blank spaces, etc.

Examples: "John", "2468", "\n", etc.

5. Boolean Literals:

Boolean literals have only two values and so are divided into two literals:

- **True** represents a real boolean value
- **False** represents a false boolean value

So, Boolean literals represent the logical value of either true or false. These values aren't case-sensitive and are equally valid if rendered in uppercase or lowercase mode. Boolean literals can also use the values of “0” and “1.”

Examples:

boolean b = true;

boolean d = false;

Final Attribute / Final Variable / Final Keyword:

- When a variable is declared as final, its value cannot be changed once it has been initialized. This is useful for declaring constants or other values that should not be modified.
- Final variables must be initialized either at the time of declaration or in the class's constructor. This ensures the variable's value is set and cannot be changed.
- The use of a final can sometimes improve **performance**, as the compiler can optimize the code more effectively when it knows that a variable or method cannot be changed.
- The final can help improve **security** by preventing malicious code from modifying sensitive data or behavior.

Syntax:

- **<final keyword> <datatype> <variable>= <value>;**

Example:

```
- final double PI = 3.14159;
```

```
public class MyClass
{
        final int MAX_VALUE = 100; // The final variable initialized at the time of declaration
        final int voterAge;
        public MyClass()
        {
                final String MY_NAME = "John Doe";
                voterAge=18; //The final variable got Initialized inside the constructor
                System.out.println(MY_NAME);
        }
        public void myMethod()
        {
                 final double PI= 3.14159;  // Initialization within a method
                System.out.println(PI);
        }
        public static void main(String arg[])
        {
                MyClass mc=new MyClass();
                System.out.println(mc.MAX_VALUE);
                System.out.println(mc.voterAge);
                mc.myMethod();
        }
}
```

STATIC VARIABLE

- A static variable is associated with a **class** rather than an **instance**. A static field is declared using the **static** keyword. Static variables are shared across all instances of a class.
- There is only **one copy of a static variable per class**. Non-static variables cannot be called inside static methods.
- If any class instance modifies a static variable's value, the change is reflected across all class instances. Static variables in Java are **memory-efficient** as they are not duplicated for each instance.

The static can be:

1.Variable (also known as a class variable)

2.Method (also known as a class method)

3.Block

4.Nested class

Syntax: <static keyword> <Data type> <Variable Name>=Value;

Ex: static String CollegeName= “SVCET”;

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

```
//Program without static variable
class Counter
{
        int count=0;//will get memory each time
```

when the instance is created

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

```
//Program with static variable
class Counter
{
        Static int count=0;
        //will get memory only once and retain its value


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

JAVA STATIC METHOD:

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can only access static data and can change its value. It cannot access non static data members and non static methods.
- Static methods can be accessed directly in non-static and static methods.

Syntax to Declare the Static Method

```
//static keyword must be used for the static method
static return_type methodname(parameter)
{
    //Code to be executed
}
```

Syntax to Call a Static Method

```
classname.methodname();
```

```
public class StaticMethod
{
        static int a = 40;    // static variable
        int b = 50;  // instance variable
        void simpleDisplay()
        {
                System.out.println(a);
                System.out.println(b);
        }
        static void staticMethod() // Declaration of a static method.
        {
                System.out.println(a);
                System.out.println(b); // non-static variable ‘b’ cannot be referenced from a static
        }
        public static void main(String[] args) // main method
        {
                StaticMethod obj = new StaticMethod();
                obj.simpleDisplay();
                staticMethod(); // Calling static method.
        }
}
```

FORMATTED OUTPUT IN JAVA USING PRINTF():

- In some times if we want the output of a program to be printed in a given **specific format**. This can be done as like as done in the ‘C’ programming with the help of printf () statement.
- Two methods can be used to format the output in Java:
  - Using the **printf( )** Method
  - Using the **format( )** Method

There are certain data types are mentioned below:

- For Number Formatting
- Formatting Decimal Numbers
- For Boolean Formatting
- For String Formatting
- For Char Formatting
- For Date and Time Formatting

i). For Number Formatting

The number itself includes Integer, Long, etc. The formatting Specifier used is %d.

Below is the implementation of the above method:

```
class ABC
{
    public static void main (String[] args)
    {
        int a=10000;
        System.out.printf("%,d",a);
    }
}
```

Output:

10,000

ii). For Decimal Number Formatting

Decimal Number Formatting can be done using print() and format specifier %f .

```
class ABC
{
    public static void main(String[] args)
    {
        double a = 3.14159265359;
        System.out.printf("%f\n", a);
        System.out.printf("%5.3f\n", a);
        System.out.printf("%5.2f\n", a);
    }
}
```

Output:

3.141593
3.142
 3.14

iii). For Boolean Formatting

Boolean Formatting can be done using printf and (‘%b’ or ‘%B’) depending upon the result needed.

```
class ABC
{       public static void main(String[] args)
        {
                int a = 10;
                Boolean b = true, c = false;
                Integer d = null;
                System.out.printf("%b\n", a);
                System.out.printf("%B\n", b);
                System.out.printf("%b\n", c);
                System.out.printf("%B\n", d);
    }   }
```

Output:

```
true
TRUE
false
FALSE
```

iv). For Char Formatting

Char Formatting is easy to understand as it need printf() and Charracter format specifier used are ‘%c’ and ‘%C’.

```
class ABC
{       public static void main(String[] args)
        {
                char c = ‘A';
                System.out.printf("%c\n", c);
                System.out.printf("%C\n", c);
    }
}
```

Output:

```
a
A
```

v). For String Formatting:

String Formatting requires the knowledge of Strings and format specifier used '%s' and '%S'.

```
class GFG {
    public static void main(String[] args)
    {       String str = "Java Programming";
            System.out.printf("%s \n", str);
            System.out.printf("%S \n", str);
            str="JAVA";
            System.out.printf("%S \n", str);
            System.out.printf("%s \n", str);
    } }
```

Output:

Java Programming
JAVA PROGRAMMING
JAVA
JAVA

vi). For Date and Time Formatting

Formatting of Date and Time is not as easy as the data-type used above. It uses more than simple format specifier.

```
import java.util.*;
class GFG {
    public static void main(String[] args)
    {
        Date time = new Date();
        System.out.printf("Current Time: %tT\n", time);
        System.out.printf("Hours: %tH  Minutes: %tM Seconds: %tS\n", time,time, time);
        System.out.printf("%1$tH:%1$tM:%1$tS %1$tp %1$tL %1$tN %1$tz %n", time);
    } }
```

Output:

Current Time: 19:41:41
Hours: 19 Minutes: 41 Seconds: 41
19:41:41 pm 815 815000000 +0000

CONTROL STATEMENTS:

In Java, control statements can be divided into the following three categories:

- Selection Statements
- Iteration Statements
- Jump Statements

SELECTION STATEMENTS

Selection statements allow you to control the flow of program execution on the basis of the outcome of an expression or state of a variable known during runtime.

Selection statements can be divided into the following categories:

- The if and if-else statements
- The if-else statements
- The if-else-if statements
- The switch statements

The if statements

The first contained statement (that can be a block) of an if statement only executes when the specified condition is true.

If the condition is false and there is not else keyword then the first contained statement will be skipped and execution continues with the rest of the program.

The condition is an expression that returns a boolean value.

Example

```
•package com.deepak.main;
•import java.util.Scanner;
•public class IfDemo
•{
•   public static void main(String[] args) {
•      int age;
•      Scanner inputDevice = new Scanner(System.in);
•      System.out.print("Please enter Age: ");
•      age = inputDevice.nextInt();
•      if(age > 18)
•         System.out.println("above 18 ");  }
•}
```

The if-else statements

In if-else statements, if the specified condition in the if statement is false, then the statement after the else keyword (that can be a block) will execute.

Example

```
import java.util.Scanner;
public class IfElseDemo
{
    public static void main( String[] args )
    {
        int age;
        Scanner inputDevice = new Scanner( System.in );
        System.out.print( "Please enter Age: " );
        age = inputDevice.nextInt();
        if ( age >= 18 )
            System.out.println( "above 18 " );
        else
            System.out.println( "below 18" );
   }
```

The if-else-if statements

Whenever the condition is true, the associated statement will be executed and the remaining conditions will be bypassed. If none of the conditions are true then the else block will execute.

Syntax:

```
if(condition)
   statements;
else if (condition)
   statements;
else if(condition)
   statement;
else
   statements;
```

```
import java.util.Scanner;
public class IfElseIfDemo
{
   public static void main( String[] args )
   {
      int age;
      Scanner inputDevice = new Scanner( System.in );
      System.out.print( "Please enter Age: " );
      age = inputDevice.nextInt();
      if ( age >= 18 && age <=35 )
         System.out.println( "between 18-35 " );
      else if(age >35 && age <=60)
         System.out.println("between 36-60");
      else
         System.out.println( "not matched" );
   }
}
```

Switch Statements:

The switch statement is a multi-way branch statement. The switch statement of Java is another selection statement that defines multiple paths of execution of a program. It provides a better alternative than a large series of if-else-if statements.

```
import java.util.Scanner;
public class SwitchDemo
{
    public static void main( String[] args )
    {
        int age;
        Scanner inputDevice = new Scanner( System.in );

        System.out.print( "Please enter Age: " );
        age = inputDevice.nextInt();
        switch ( age )
        {
            case 18:
                System.out.println( "age 18" );
                break;
            case 19:
                System.out.println( "age 19" );
                break;
            default:
                System.out.println( "not matched" );
                break;
        }
    }
}
```

ITERATIONSTATEMENTS

Repeating the same code fragment several times until a specified condition is satisfied is called iteration.

Iteration statements execute the same set of instructions until a termination condition is met.

Java provides the following loop for iteration statements:

The while loop
The for loop
The do-while loop

The while loop

It continually executes a statement (that is usually be a block) while a condition is true. The condition must return a boolean value.

Example

```
public class WhileDemo
{
    public static void main( String[] args )
    {
        int i = 0;
        while ( i < 5 )
        {  System.out.println( "Value :: " + i );
            i++;
        }
    }
}
```

do-while:

- The Java *do-while loop* is executed at least once because condition is checked after loop body.
- If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

Syntax:

```
do{
//code to be executed
}while(condition);
```

```
public class DoWhileExample
{
    public static void main(String[] args)
    {
        int i=1;
        do
        {
            System.out.println(i);
            i++;
        }while(i<=10);
    }
}
```

For loop:
The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

1.Initialization: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2.Condition: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3.Statement: The statement of the loop is executed each time until the second condition is false.
4.Increment/Decrement: It increments or decrements the variable value. It is an optional condition.

Syntax:

```
for(initialization;condition;incr/decr)
{
   //statement or code to be executed
}
```

```
public class ForExample
{
   public static void main(String[] args)
   {
      for(int i=1;i<=10;i++)
      {   System.out.println(i);  }
   }
}
```

JUMP STATEMENTS:

Java supports three jump statement: **break, continue** and **return**. These three statements transfer control to other part of the program.

Break: In Java, break is majorly used for:

1. Terminate a sequence in a switch statement (discussed above).
2. To exit a loop.

Using break to exit a Loop

1.Using break, we can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of the loop.

Note: Break, when used inside a set of nested loops, will only break out of the innermost loop.

```
class BreakLoopDemo
{
    public static void main(String args[])
    {
        for (int i = 0; i < 10; i++)
        {
            if (i == 5)
                break;
              System.out.println("i: " + i);
        }
        System.out.println("Loop complete.");
    }
}
```

Continue: Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. This is, in effect, a goto just past the body of the loop, to the loop's end. The continue statement performs such an action.

```
class ContinueDemo
{  public static void main(String args[])
   {
      for (int i = 0; i < 10; i++)
      {
         // If the number is even skip and continue
         if (i%2 == 0)
            continue;
         // If number is odd, print it
         System.out.print(i + " ");
      }
   }
}
```

THE Return Statement:

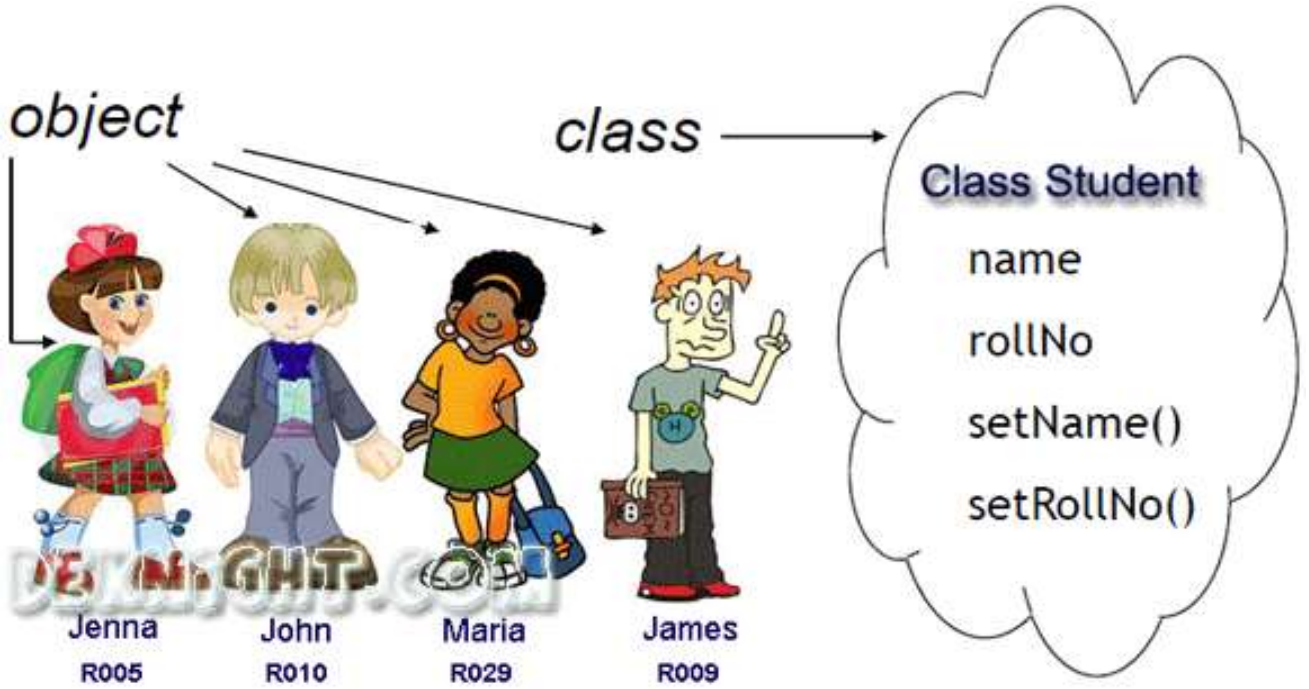
This statement is mainly used in methods in order to terminate a method in between and return back to the caller method. It is an optional statement. That is, even if a method doesn't include a return statement, control returns back to the caller method after execution of the method.

CLASS:

- A class is a group of objects that has common properties. It is a template or blueprint from which objects are created.
- **Data Members:** Attributes are variables that store data within an object. They can be of various data types (e.g., primitive or reference).
- **Methods:** Methods define the actions that an object can perform. They take inputs (parameters) and return outputs (return values).

A class in java can contain:

- **data member**
- **method**
- **constructor**
- **block**
- **class and interface**



Class Declaration:

Syntax:

```
class <class_name>
{
   data members;
   methods;
}
```

Example:

```
class Student
{
        int rollNo; //class attributes
        String name; //class attributes
        void display() // Method
        {
                //Method Statements
        }
}
```

- **class keyword :** Indicates that a class is being defined.
- **ClassName :** The name of the class, following Java naming conventions (e.g., Student).
- **Body :** Enclosed within curly braces, contains the class's attributes (fields) and methods.

Access Control for class:

• **Default:** Classes declared without any access modifier are considered package-private by default. They are accessible only within the same package.

• **Public:** A public class can be accessed anywhere in the program, including other packages. This is commonly used for classes that are part of a public API.

• **No private classes:** Java does not allow classes to be declared as private. This is because classes are generally intended to be used by other classes, and making them private would severely limit their usefulness.

Example:

```
// class MyClass
{
        // ...
}
public class MyPublicClass
{
        // ...
}
```

OBJECT:

Object is an instance of a class. An entity that has state and behavior is known as an object e.g. chair, bike, marker, pen, table, car etc. It can be physical or logical (tengible and intengible). The example of integible object is banking system.

An object has three characteristics:

- **state:** represents data (value) of an object.
- **behavior:** represents the behavior (functionality) of an object such as deposit, withdraw etc.
- **identity:** Object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. But, it is used internally by the JVM to identify each object uniquely.

For Example: Pen is an object. Its name is Reynolds, color is white etc. known as its state. It is used to write, so writing is its behavior.

Syntax:

<class name> <object name> = new <class name()>;

Example:

Student s = new Student();

Example Prog. for class and Object:

```
public class MyClass
{
    // Method to display a message
    public void displayMessage()
    {
        System.out.println("Hello from a method!");
    }
    // Method to calculate the sum of two numbers
    public int calculateSum(int a, int b)
    {
        return a + b;
    }
public static void main(String[] args)
{
    // Create an object of the MyClass
    MyClass obj = new MyClass();
    // Calling methods using object
    obj.displayMessage();
    int num1 = 10, num2 = 20;
    int sum = obj.calculateSum(num1, num2);
    System.out.println("Sum of  num1&num2= " + sum);
 }
 }
```

Output:

Hello from a method
Sum of num1&num2= 30

ASSIGNING ONE OBJECT TO ANOTHER:

- When you assign one object to another, you're essentially creating a reference to the original object. Both variables will point to the same memory location, meaning any changes made to one object will also affect the other.

```
class Person
{
    String name;
    int age;
}
public class Main
{   public static void main(String[] args)
    {
        Person person1 = new Person();
        person1.name = "Alice";
        person1.age = 30;
        System.out.println(person1.name);
        System.out.println(person1.age);
        Person person2 = person1; //Assigning one object to another
        System.out.println(person2.name);
        System.out.println(person2.age);
        // Modifying person1 will also affect person2
        person1.name = "Bob";
        person1.age = 25;
        System.out.println(person2.name);
        System.out.println(person2.age);
    }
}
```

Output:

```
Alice
30
Alice
30
Bob
25
```

Access Control for Class Members:

- **Access Modifiers:** Control the visibility of the classes, attributes, and methods for other classes.
- **public:** Accessible from anywhere in the program.
- **private:** Accessible only within the same class.
- **protected:** Accessible within the same package and subclasses.
- **default (package-private):** Accessible within the same package.

```
public class MyClass
{
    private int myPrivateField;
    protected int myProtectedField;
    public int myPublicField;
    int myDefaultField;
    public void myPublicMethod()
    {
        // ...
    }
    private void myPrivateMethod()
    {
        // ...
    }
    protected void myProtectedMethod()
    {
        // ...
     }
     void myDefaultMethod()
     {       // …       }
}
```

```
class Modifier
{
   private int privatevar=1;
   public int publicvar=2;
   protected int protectedvar=3;
   int defaultvar=4;
   public void publicMethod()
   {
      System.out.println("Public method");
      privateMethod();
   }
   private void privateMethod()
   {
      System.out.println("Private method");
      System.out.println(privatevar);
      System.out.println(publicvar);
      System.out.println(protectedvar);
      System.out.println(defaultvar);
   }
   protected void protectedMethod()
   {
      System.out.println("Protected method");
   }
   void defaultMethod()
   {
      System.out.println("Default method");
   }
}
class Main
{
   public static void main(String arg[])
   {
      Modifier M=new Modifier();
      //M.privateMethod(); - Private method
                cant access outside of the class
      M.publicMethod();
      M.protectedMethod();
      M.defaultMethod();
   }
}
```

Accessing Private Members of a Class in Java:

In Java, private members of a class are accessible within the same class. This ensures data encapsulation and prevents unauthorized access. However, there are a few ways to access private members indirectly:

1. Public Methods:

•The most common and recommended approach is to provide public methods that expose the functionality of the private members. These methods can perform validation, calculations, or other necessary operations before accessing or modifying the private data.

```
class Person
{
   private String name;
        private int age;
        public void getDetails()
        {
                name="Ram";
                age=20;
        }
        public void printDetails()
        {
                System.out.println(name);
                System.out.println(age);
        }
        public static void main(String arg[])
        {       Person p=new Person();
                p.getDetails();
                p.printDetails();
        }
}
```

```
Output:
        Ram
        20
```

2. Inner Classes:

• Inner classes defined within a class have access to the outer class's private members. This can be useful for implementing helper methods or data structures that are closely related to the outer class.

```
class Person
{
    private String name="Private variable";
    private int age=19;
    class InnerClass
    {
        public void printDetails()
        {
            System.out.println("Name: " + name);
            System.out.println("Age: " + age);
        }
    }
}
```

```
class Main
{
    public static void main(String arg[])
    {
        Person.InnerClass ic=new Person().new InnerClass();
        ic.printDetails();
    }
}
```

Output:

```
Name: Private variable
Age: 19
```

CONSTRUCTOR:

- A constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.
- It is a special type of method which is used to initialize the object.
- Every time an object is created using the new() keyword, at least one constructor is called.
- It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructors:

There are two rules defined for the constructor:

1. Constructor name must be the **same as its class name.**
2. A Constructor must have **no explicit return type.**

Note:

- A Java constructor cannot be abstract, static, final, and synchronized
- We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public, or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor).
2. Parameterized constructor.

Default Constructor (no-arg constructor):

- **Automatic Creation:** A default constructor is a special type of constructor that is automatically generated by the Java compiler when a class doesn't explicitly define any constructors.
- **No Arguments:** It takes no arguments and has an empty body.
- **Initialization:** The default constructor initializes the instance variables of the class to their default values.

Default Values:

- **Primitive Types:** The default values for primitive data types are as follows:
  - int: 0
  - double: 0.0
  - float: 0.0f
  - long: 0L
  - char: '\u0000' (null character)
  - boolean: false
  - byte: 0
  - short: 0
- **Reference Types:** The default value for reference types is null.

Syntax for default constructor:

```
class classname
{
        <classname>()
        {
                //…
        }
}
```

```
public class ConEx
{
   int  id;
   String name;
   // Default Constructor - no arguments
   public ConEx()
   {
            id = 10;
            name = "Svcet";
            System.out.println("Default constructor called");
   }
   public void display()
   {
            System.out.println("Id: " + id);
            System.out.println("Name: " + name);
   }
   public static void main(String[] args)
   {
            // Creating objects using different constructors
            ConEx obj1 = new ConEx();
            obj1.display();
     }
}
```

Output:

```
Default constructor called
Id: 10
Name: Svcet
```

Parameterized Constructors:

• **Initialization with Arguments:** Parameterized constructors are used to initialize object attributes with specific values provided as arguments during object creation.

• Parameterized constructors are used to create user instances of objects with user defined states.

• **Multiple Constructors:** A class can have multiple parameterized constructors, each with different parameters and initialization logic.

• **Overloading:** Parameterized constructors can be overloaded, meaning they can have the same name but different parameter lists. The compiler determines which constructor to call based on the arguments provided during o

Syntax for parameterized constructor:

```
class classname
{
        <classname>(Parameter list)
        {
                //…
        }
}
```

```
public class ConEx
{
        int id;
        String name;
        // Parameterized Constructor
        public ConEx(int i, String n)
        {
            id = i;
            name = n;
            System.out.println("Parameterized constructor called");
        }
        public void display()
        {
                System.out.println("Id: " + id);
                System.out.println("Name: " + name);
        }
```

```
public static void main(String[] args)
{
    // Creating objects using different constructors
    ConEx obj2 = new ConEx(20, "Svcet-Chittoor”);
    obj2.display();
 }
}
```

OVERLOADED CONSTRUCTORS:

•**Multiple Constructors with Same Name:** Overloaded constructors are multiple constructors within the same class that have the same name but different parameter lists.

•**Parameter Types and Number:** The compiler determines which overloaded constructor to call based on the types and number of arguments provided during object creation.

•**Return Type:** Constructors do not have a return type.

•**Purpose:** Overloaded constructors provide flexibility in object creation, allowing you to initialize objects with different sets of values.

Syntax:

```
public class MyClass
{
        public MyClass()
        {       // ...
        }
        public MyClass(int x)
        {       // ...
        }
        public MyClass(String y, double z)
        {       // ...
        }
}
```

```
public class Main
{       String Name;
        int i;
        double PI;
        Main()
        {       System.out.println("Control from Default Constructor");
        }
        Main(int i)
        {       this.i=i;
                System.out.println("Control from integer parameter Constructor "+ i );
        }
        Main(String Name, double PI)
        {       this.Name=Name;
                this.PI=PI;
                System.out.println("Control from two-parameter Constructor "+Name+" "+PI);
        }
        public static void main(String[] args)
        {       Main obj1 = new Main();
                Main obj2 = new Main(10);
                Main obj3 = new Main("Hello",3.14);
}       }
```

Output:

```
Control from Default Constructor
Control from integer parameter Constructor 10
Control from two-parameter Constructor Hello 3.14
```

Nested Classes:

•**Classes Within Classes:** Nested classes are classes defined within the scope of another class. They can be either static or non-static.

•**Accessibility:**

- **Static Nested Classes:** Can be accessed directly using the outer class's name. They don't have access to the outer class's instance variables or methods unless they are passed as arguments.
- **Non-Static Nested Classes (Inner Classes):** Can only be accessed from within an instance of the outer class. They have direct access to the outer class's instance variables and methods.

Syntax:

```
public class OuterClass
{
        // Static nested class
        public static class StaticNestedClass (inner class)
        {
                // ...
        }
        // Non-static nested class (inner class)
        public class InnerClass
        {
                // ...
}       }
```

```
public class OuterClass
{       private int outerField;
        public class InnerClass
        {       public void printOuterField()
                {
                        System.out.println("Outer field: " + outerField);
        }       }
        public static class StaticNestedClass
        {       public static void printMessage()
                {
                        System.out.println("Static nested class message");
                }
}       }
class Main
{       public static void main(String arg[])
        {
                OuterClass.InnerClass oc=new OuterClass().new InnerClass();
                oc.printOuterField();
                OuterClass.StaticNestedClass.printMessage();
        }
}
```

Output:

Outer field : 0

Static nested class message

FINAL CLASS:

When a class is declared with the **final** keyword, it is called a **final class**. A final class cannot be **extended(inherited)**.

There are two uses of a final class:

1: One is to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float, etc. are final classes. We can not extend them.

2: The other use of final with classes is to create an immutable class(once an object is created, we cannot change its content) like the predefined String class. One can not make a class immutable without making it final.

Syntax:

```
final class A
{
        // methods and fields
}
// The following class is illegal – because final class cannot be inherited
class B extends A
{
        // COMPILE-ERROR! Can't subclass A
}
```

```
public final class FinalClass
{
        public void display(int num)
        {
            System.out.println(“Sup-Int argument:" + num);
        }
}
public class DerievedClass extends FinalClass // Compile Error – Finale class cannot be inherited
{
        public void display(int num)
        {
            System.out.println(“Sub-Int argument:" + num);
        }
}
```

PASSING BY VALUE:

• **Primitive Types:** When primitive data types (e.g., int, double, boolean) are passed as arguments to methods, their values are copied to the method's local variables. Any modifications made to these local variables within the method do not affect the original values outside the method.

Example :

```
public class PassByValue
{
        public static void main(String[] args)
        {
                int x = 10;
                System.out.println(x);
                modifyInt(x);
                System.out.println(x);
        }
        public static void modifyInt(int num)
        {
                num = 20;
                 System.out.println(num);
        }
}
```

Output:
10
20
10

PASSING BY REFERENCE:

• **Object References:** When objects are passed as arguments to methods, their references are copied. This means that both the original object and the method's local variable reference the same object in memory. Any modifications made to the object within the method will be reflected in the original object outside the method.

Example:

```
public class Main
{
   String name;    int age;
   Main(String name, int age)
   {
      this.name=name;
      this.age=age;
   }
   public static void modifyReference(Main m)
   {
         m.modifyDetails(“Govind",25);
   }
   void modifyDetails(String n, int a)
   {
         name=n;
         age=a;
   }
   public static void main(String[] args)
   {
         Main main = new Main(“Ram", 29);

         System.out.println(main.name);
         System.out.println(main.age);

         modifyReference(main);

         System.out.println(main.name);
         System.out.println(main.age);
   }
}
```

```
Output:  Ram
         29
         Govind
         25
```

THIS KEYWORD:

- **“this”** is a reference variable that refers to the current object.
- When we are going for **“this”**, for example, if we have a defined parameterized constructor whose formal parameters name is similar to the class data member.
- Then **JVM** will go into confusion about which variable is the current class variable.
- So simply we apply **"this"** keyword. It generates the difference between the current class data member and to formal parameter.

```
class Test
{       int a;
        int b;
        Test(int a, int b)
        {       this.a = a;
                this.b = b;
        }
        void display()
        {       System.out.println("a = " + a + "  b = " + b);
        }
        public static void main(String[] args)
        {     Test object = new Test(10, 20);
              object.display();
        }
}
```

Output:

a = 10
b = 20

METHODS:

Introduction:

- **Code Blocks:** Methods are blocks of code that perform specific tasks within a class. They encapsulate functionality and can be reused multiple times.
- **Parameters and Return Values:** Methods can take input parameters and return output values.
- **Accessibility Modifiers:** Methods can have different access modifiers (public, private, protected, default) to control their visibility.
- **Static Methods:** Static methods belong to the class itself and can be called without creating an instance of the class.
- **Instance Methods:** Instance methods belong to individual objects and can only be called on specific instances.

Naming a Method:

- While defining a method, remember that the method name must start with a **lowercase** letter.
- If the method name has more than two words, the first letter of each word must be in **uppercase** except the first word.

For example:

- **Single-word method name:** sum(), area()
- **Multi-word method name:** areaOfCircle(), stringComparision()

There are two types of methods in Java:

1. Predefined Method:

- predefined methods are the methods that are already defined in the Java class libraries are known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point.

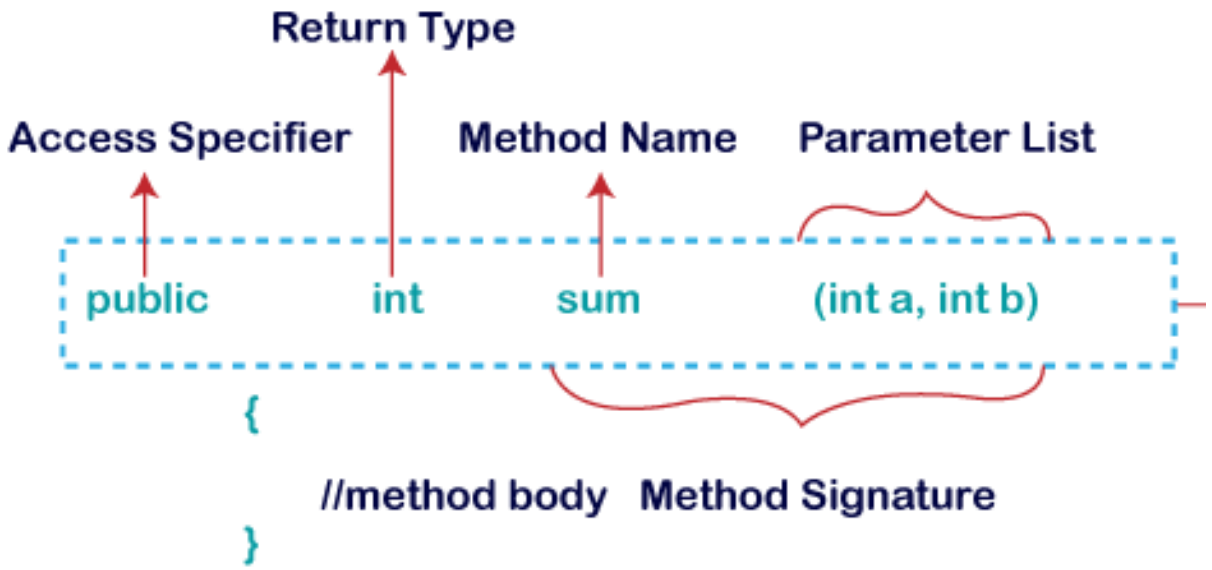
2. User-defined Method:

- The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

Method Declaration:

Syntax:

```
public class MyClass
{           // Instance method
            public void myInstanceMethod(int x, String y)
            {
                        // ... return value;
            }
            // Static method
            public static int myStaticMethod(double z)
            {
                        // ... return result;
}           }
```



Example Program for method defining and invoking :

```
public class MyClass
{
    // Method to display a message
    public void displayMessage()
    {
        System.out.println("I am Ordinanry method!");
    }
    // static Method
    public static void display()
    {
        System.out.println("I am static method!");
    }
    public static void main(String[] args)
    {
        // Create an object of the MyClass
        MyClass obj = new MyClass();
        // Calling or invoking method using object
        obj.displayMessage();
        // Invoking static method without object
        display();
    }
}
```

Output:

I am Ordinary method
I am static method

OVERLOADED METHODS – METHOD OVERLOADING:

- **Multiple Methods with Same Name:** Method overloading allows you to define multiple methods within the same class with the same name but different parameter lists.
- **Parameter Types:** The compiler determines which overloaded method to call based on the types and number of arguments passed during the method invocation.
- **Return Type:** The return type of the overloaded methods is not considered for overloading.

Syntax:

```
public class MyClass
{
        public void myMethod(int x)
        { // ... }
        public void myMethod(String y)
        { // ... }
        public int myMethod(double z)
        { // ... return result; }
}
```

```
public class MethodOverloading
{      // Display method without parameter
       public void display()
       {
            System.out.println("No parameters");
       }
       // Display method with one integer parameter
       public void display(int num)
       {
           System.out.println("Integer argument: " + num);
       }
       // Display method with one string parameter
       public void display(String str)
       {
           System.out.println("String argument: " + str);
       }
public static void main(String[] args)
{
    MethodOverloading obj = new MethodOverloading();
    obj.display(); // invokes display method without parameter
    obj.display(10); // invokes display method with integer
                          // parameter
     obj.display("Hello"); // invokes display method with string
                                  // parameter
 }
}
```

Output:
No Parameters
Integer argument: 10
String argument: Hello

METHOD OVERRIDING:

•**Subclasses Redefining Methods:** Method overriding occurs when a subclass provides a new implementation for a method inherited from its superclass.

•**Same Signature:** The overriding method must have the same name, return type, and parameter list as the original method in the superclass.

•**Access Modifiers:** The overriding method can have a more accessible access modifier (e.g., public instead of protected) but not a less accessible one.

•**Dynamic Binding:** The method that is actually called at runtime is determined dynamically based on the object's actual type, a process known as dynamic binding or late binding.

Syntax:

```
class SuperClass
{
    public void myMethod()
   { // ... }
}
class SubClass extends SuperClass
{
    @Override public void myMethod()
    { // ... }
}
```

```
public class SuperClass
{
        public void display(int num)
        {
            System.out.println(“Sup-Int argument:" + num);
        }
}
public class DerievedClass extends BaseClass
{
        public void display(int num)
        {
            System.out.println(“Sub-Int argument:" + num);
        }
}
```

```
class main
{
  public static void main(String[] args)
  {
     SuperClass obj = new SuperClass();
     obj.display(10); // invokes SuperClass method
     SuperClass obj = new DerievedClass();
     obj.display(20); // invokes SubClass method
   }
}
```

Output:

Sup-Int argument: 10

Sub-Int argument: 20

FINAL METHOD:

- When a method is declared with a **final** keyword, it is called a **final method.**
- A final method cannot be overridden, ie. The final method cannot be inherited.
- The Object class does this—a number of its methods are final. We must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes.

Syntax:

```
class SuperClass
{
    final <ReturnType><MethodName>()
    { // ... }
}
class SubClass extends SuperClass
{
    @Override Final Method()
    { // ... }
}
```

```
class A
{
        final void m1()
        {
                System.out.println("This is a final method.");
}       }
class B extends A
{
        void m1() // Compile-error! We can not override the final method
        {
                System.out.println("Illegal!");
}       }
```

RECURSIVE METHODS:

- **Self-Calling Methods:** Recursive methods are methods that call themselves directly or indirectly. They are often used to solve problems that can be broken down into smaller, similar subproblems.
- **Base Case:** A recursive method must have a base case, which is a condition that stops the recursion and prevents an infinite loop.
- **Recursive Case:** The recursive case is the part of the method that calls itself with a modified input.

Syntax:

```
public class RecursiveExample
{
        public static int factorial(int n)
        {
                if (n == 0)
                {
                        return 1; // Base case
                }
                else
                {
                        return n * factorial(n - 1); // Recursive case
                }
        }
}
```

```
public class Main
{
        public static void main(String[] args)
        {
            int result = RecursiveExample.factorial(5);
            System.out.println("Factorial of 5: " + result);
        }
 }
```

Output: Factorial of 5: 120

NESTING OF METHODS:

Java doesn't directly support the nesting of methods within other methods, there are a few alternative approaches that can achieve similar functionality:

1. Inner Classes:

•**Nested Classes:** Inner classes are classes defined within the scope of another class. They can be either static or non-static.

•**Method-Level Inner Classes:** You can define an inner class within a method to encapsulate related functionality and access the method's local variables.

2. Lambda Expressions:

- Lambda expressions basically express instances of functional interfaces (An interface with a single abstract method is called a functional interface. An example is java.lang.Runnable).
- Lambda expressions implement the only abstract method and therefore implement functional interfaces.

3. Using anonymous subclasses:

- It is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain “extras” such as overloading methods of a class or interface, without having to actually subclass a class.

1. INNER CLASSES:

```
class OuterClass
{
    public void outerMethod()
    {
        class InnerClass
        {
            public void innerMethod()
            {
                System.out.println("nested method");
            }
        }
        InnerClass inner = new InnerClass();
        inner.innerMethod();
        System.out.println("outer method");
    }
}
```

```
class Main

{
        public static void main(String ar[])
        {
                OuterClass oc=new OuterClass();
                        Output:
                oc.outerMethod();
        }
}
```

2. LAMBDA EXPRESSIONS:

```
public class LambExpr {
    interface myInterface {
        void run();
    }
     // function has implemented another function
    // run() using Lambda expression
    static void Foo()
    {
        // Lambda expression
        myInterface r = () ->
        {
            System.out.println("geeksforgeeks");
        };
        r.run();
    }
    public static void main(String[] args)
    {
        Foo();
}   }
```

3. USING ANONYMOUS SUBCLASSES:

```
public class Hello
{       interface myInterface
        {
                //create a local interface with one abstract method run()
                void run();
        } // function have implements another function run()
        static void Foo() //implement run method inside Foo() function.
        {
                myInterface r = new myInterface()
                {
                        public void run()
                        {       System.out.println("Hello");
                };      };
                r.run();
        }
        public static void main(String[] args)
        {
                Foo();
}       }
```

The following topics are repeated in Unit 2. These topics can be referred to - as mentioned in the following tabular column.

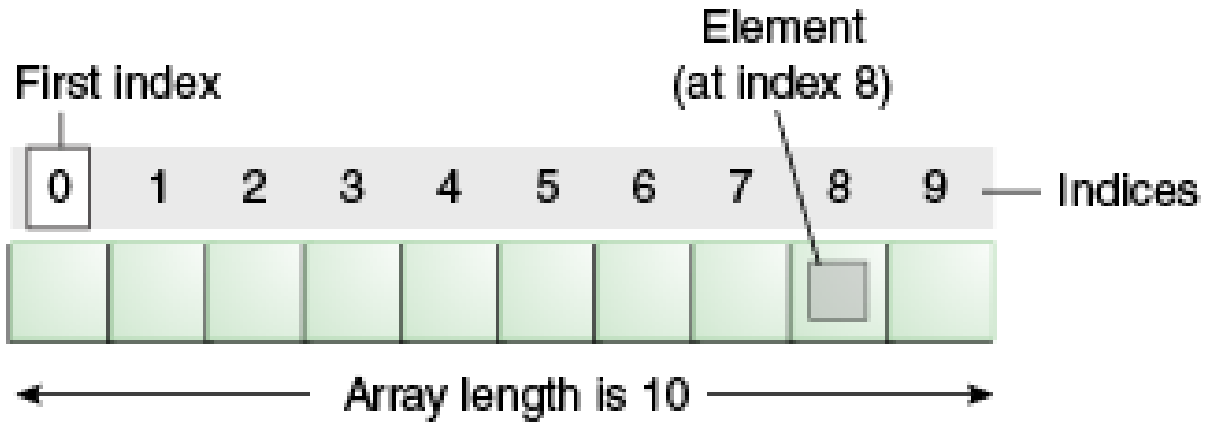
Topics	Reference
Overloaded Constructor Methods	Page 16,17 – Unit II
Class Objects as Parameters in Methods	Page 23 – Unit II
Access Control	Page 7,8 – Unit II
Attributes Final and Static	Unit I

ARRAY:

Normally, array is a collection of similar type of elements that have contiguous memory location.

Java array is an object that contains elements of similar data type. It is a data structure where we store similar elements. We can store only fixed set of elements in a java array.

Array in java is index based, first element of the array is stored at 0 index.



Advantage of Java Array

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data easily.
- **Random access:** We can get any data located at any index position.

Disadvantage of Java Array

- **Size Limit:** We can store only fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in java.

Three types of array.

- Single Dimensional Array
- Two Dimensional Array
- Three Dimensional Array

Single Dimensional Array in java

Syntax to Declare an Array in java

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Instantiation of an Array in java:

arrayRefVar[]=**new** datatype[size];

Example of single dimensional java array:

Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.

```
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output: 10 20 70 40 50

Multidimensional array in java:

In such case, data is stored in row and column based index (also known as matrix form).

Syntax to Declare Multidimensional Array in java:

```
dataType[][] arrayRefVar; (or)//  int[][] a;
dataType [][]arrayRefVar; (or)//  int [][]a;
dataType arrayRefVar[][]; (or)//  char a[][];
dataType []arrayRefVar[];   // float []a[];
```

Example to instantiate Multidimensional Array in java

```
int[][] arr=new int[3][3];//3 row and 3 column
```

Example to initialize Multidimensional Array in java:

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Example of Multidimensional java array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

```
class Testarray3
{
        public static void main(String args[])
        {
                //declaring and initializing 2D array
                int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
                //printing 2D array
                for(int i=0;i<3;i++)
                {
                        for(int j=0;j<3;j++)
                        {
                                System.out.print(arr[i][j]+" ");
                        }
                        System.out.println();
                }
        }
}
```

Output:
```
1 2 3
2 4 5
4 4 5
```

Search for Values in Arrays (Binary search)

```
class BinarySearch
{
    int binarySearch(int arr[], int l, int r, int x)
    {
        while (l <= r)
        {
            int mid = (l + r) / 2;
            if (arr[mid] == x)
            {
                return mid;
            }
            else if (arr[mid] > x)
            {
                r = mid - 1;
            }
            else
            {
                l = mid + 1;
            }
        }
        return -1;
    }
```

```
 public static void main(String args[])
{
    BinarySearch ob = new BinarySearch();
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = arr.length;
    int x = 10;
    int result = ob.binarySearch(arr, 0, n - 1, x);
    if (result == -1)
        System.out.println("Element not present");
    else
        System.out.println("Element found at index " + result);
 }
}
```

Output:

Element found at index 3

Sorting of Arrays (Bubble Sort)

```
class BubbleSort
{
   void bubbleSort(int arr[])
   {
      int n = arr.length;
      for (int i = 0; i < n - 1; i++)
        for (int j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1])
          {
              int temp = arr[j];
              arr[j] = arr[j + 1];
              arr[j + 1] = temp;
           }
   }
   void printArray(int arr[])
   {
      int n = arr.length;
      for (int i = 0; i < n; ++i)
            System.out.print(arr[i] + " ");
            System.out.println();
   }
```

```
   public static void main(String args[])
   {
       BubbleSort ob = new BubbleSort();
       int arr[] = { 64, 34, 25, 12, 22, 11, 90 };
       ob.bubbleSort(arr);
       System.out.println("Sorted array");
       ob.printArray(arr);
   }
}
```

Output:

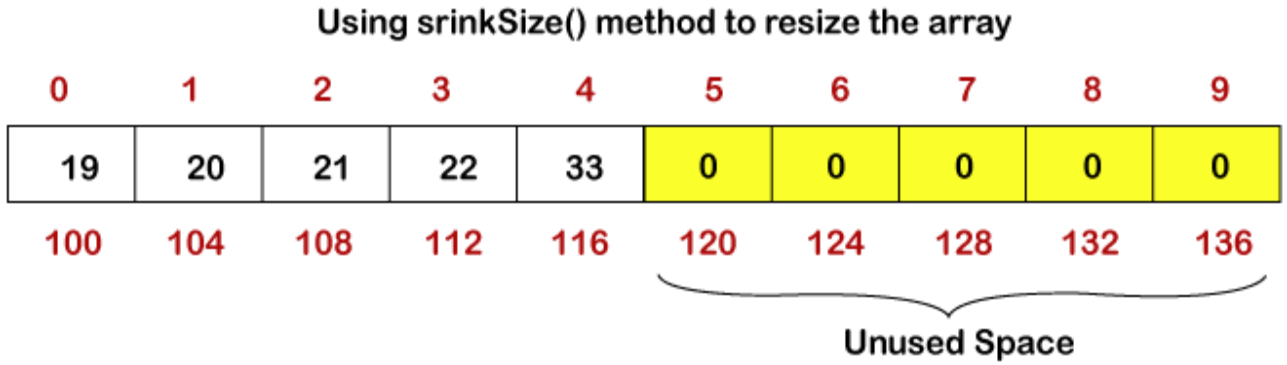
```
Sorted array
11 12 22 25 34 64 90
```

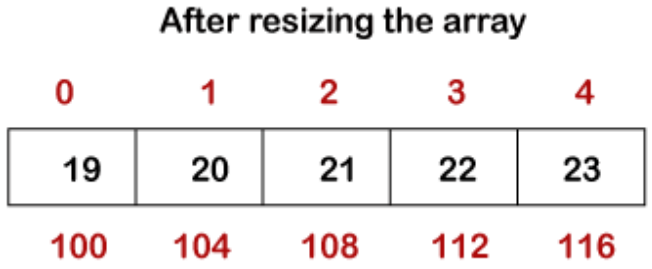
Dynamic change in Array Size:

The dynamic array is a **variable-size** list data structure. It grows automatically when we try to insert an element if there is no more space left for the new element. It allows us to add and remove elements. It allocates memory at run time using the heap. It can change its size during run time.

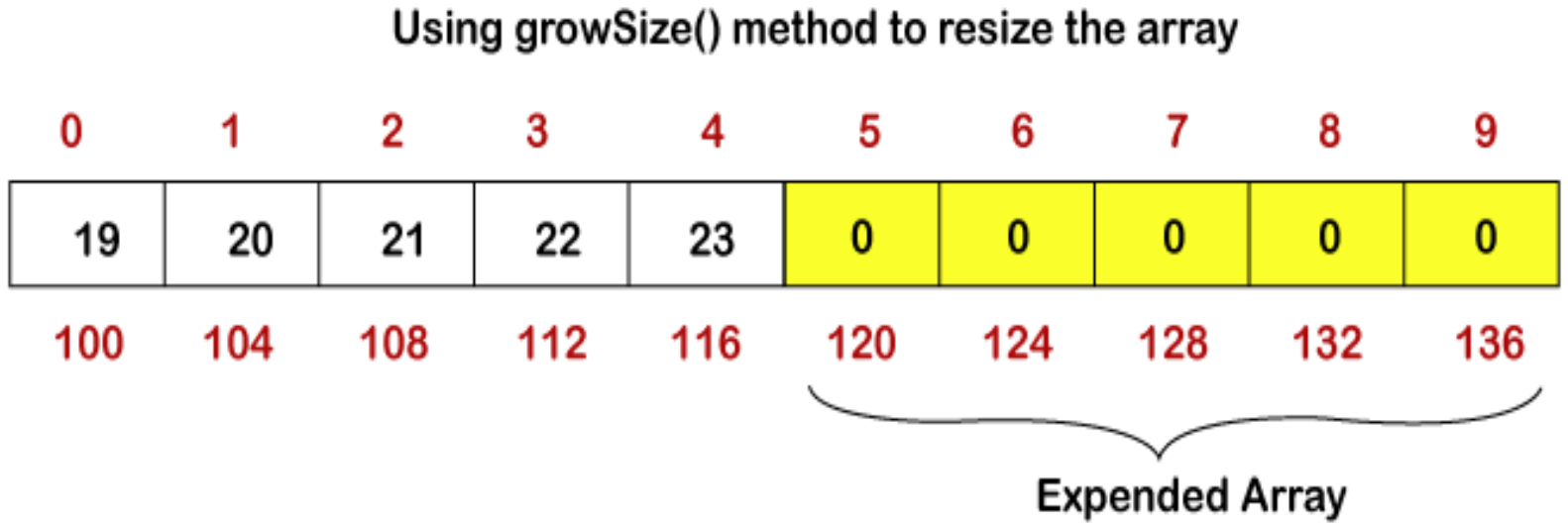
Resizing a Dynamic Array in Java:

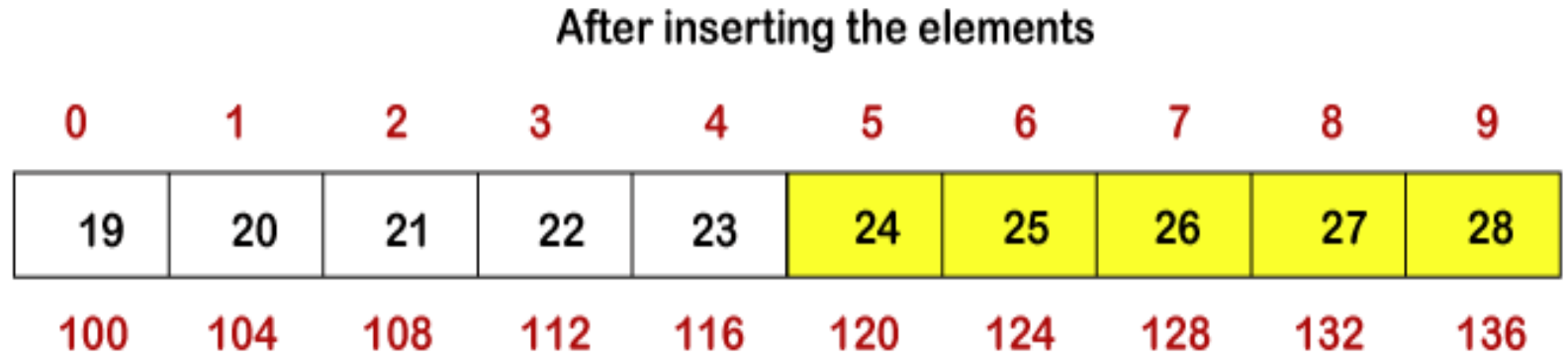
- We need to resize an array in the following two scenarios if:
- The array uses extra memory than required.
- The array occupies all the memory and we need to add elements.
- In the first case, we use the **srinkSize()** method to resize the array. It reduces the size of the array. It free up the extra or unused memory. In the second case, we use the **growSize()** method to resize the array. It increases the size of the array.
- It is an expensive operation because it requires a bigger array and copies all the elements from the previous array after that, it return the new array.





Suppose in the above array, it is required to add six more elements and, in the array, no more memory is left to store elements. In such cases, we grow the array using the **growSize()** method.





// Java Program to Implement a Dynamic Array

```
class Array
{
    private int arr[];
    private int count;
    public Array(int size)
    {
          arr = new int[size];
    }
    public void printArray()
    {
        for (int i = 0; i < count; i++)
            System.out.print(arr[i] + " ");
    }
    public void insert(int ele)
    {
            if (arr.length == count)
            {
                int newArr[] = new int[2 * count];
                for (int i = 0; i < count; i++)
                    newArr[i] = arr[i];
                arr = newArr;
            }
            arr[count++] = ele;
        }
    }
    public class Main
    {
        public static void main(String[] args)
        {    Array numbers = new Array(3);
             numbers.insert(10);
             numbers.insert(20);
             numbers.insert(30);
             numbers.insert(50);
             numbers.printArray();
        }
    }
```

Output `10 20 30 50`

INHERITANCE:

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship.

Why use inheritance in java?

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Important terminology:

Super Class: The class with inherited features is known as a super class(a base class or a parent class).

Sub Class: The class that inherits the other class is known as a sub class(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.

Reusability: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

How to use inheritance in Java

The keyword used for inheritance is **extends**.

Syntax :

```
class derivedclass extends baseclass
    {
        //methods and fields
    }
```

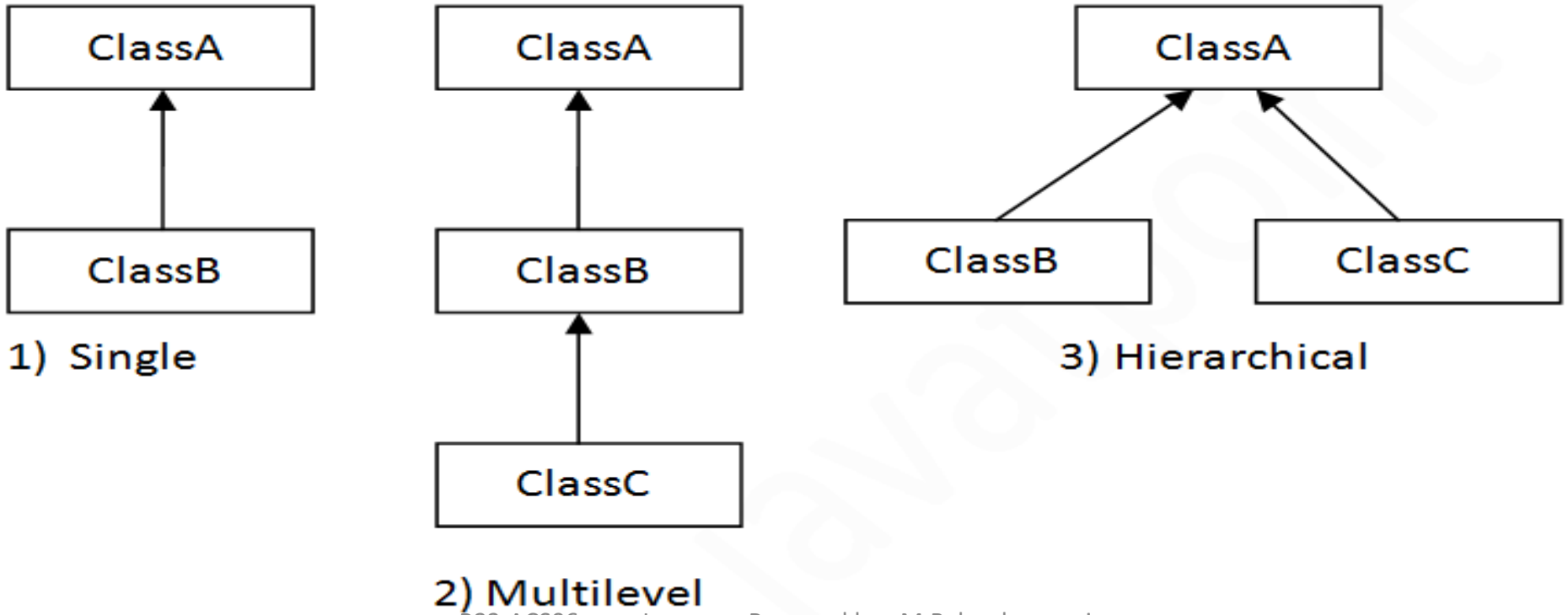
NOTE: The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

```
class Employee
{
        float salary=40000;
}
class Programmer extends Employee
{
        int bonus=10000;
        public static void main(String args[])
        {
                Programmer p=new Programmer();
                System.out.println("Programmer salary is:"+p.salary);
                System.out.println("Bonus of Programmer is:"+p.bonus);
        }
}
```

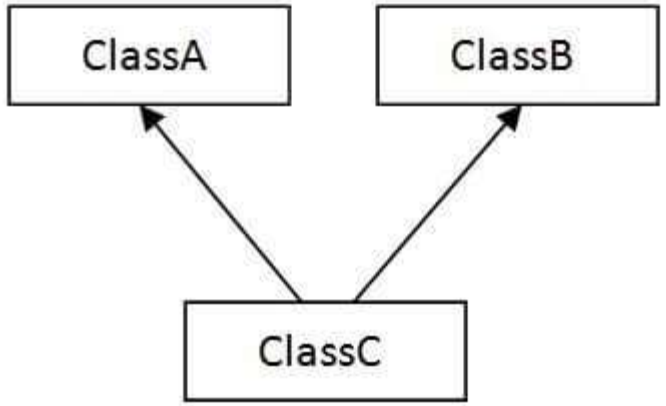
Types of inheritance in Java:

Based on **class** in Java – it supports three types of inheritance:

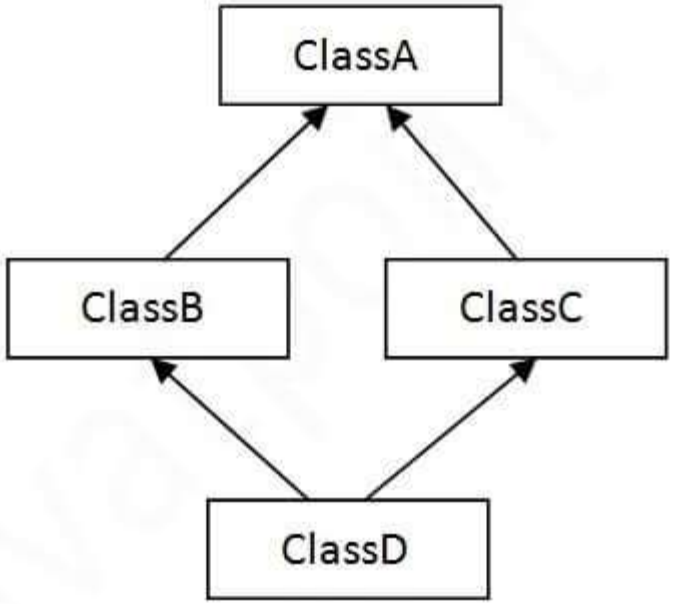
1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance.



Note: In Java programming, multiple and hybrid inheritance are supported only through the interface.



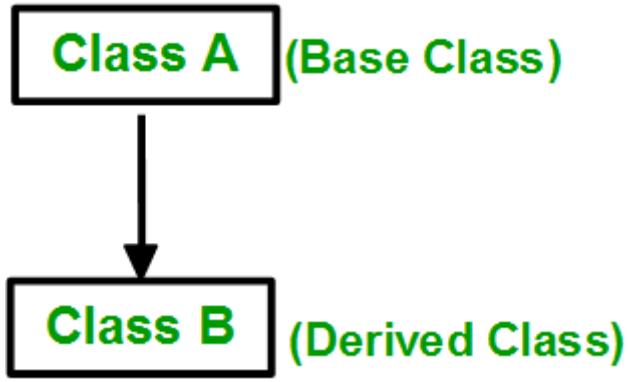
4) Multiple



5) Hybrid

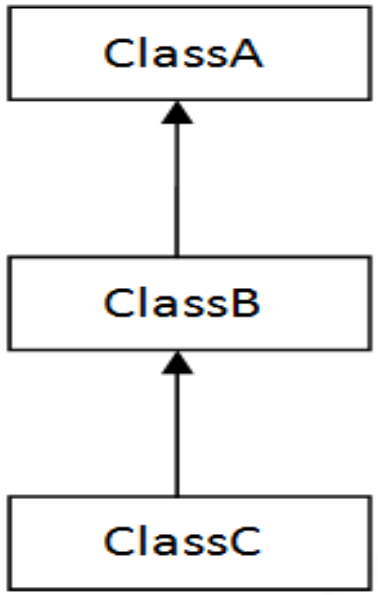
Single Inheritance: In single inheritance, a class is allowed to inherit from only one class.
i.e. one sub class is inherited by one base class only.

```
class Shape
{       int length;
        int breadth;
}
public class Rectangle extends Shape
{       int area;
        public void calcualteArea()
        {       area = length*breadth;
        }
        public static void main(String args[])
        {
                Rectangle r = new Rectangle();
                r.length = 10;
                r.breadth = 20;
                r.calcualteArea();
                System.out.println("The Area of rectangle is="+r.area);
        }
}
```



Multilevel Inheritance :

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

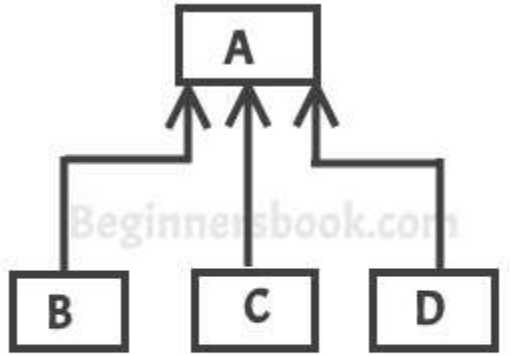


2) Multilevel

```
class Shape
{       public void display()
        {       System.out.println("Inside display");
        }
}
class Rectangle extends Shape
{       public void area()
        {       System.out.println("Inside area");
        }
}
class Cube extends Rectangle
{       public void volume()
        {       System.out.println("Inside volume");
        }
        public static void main(String[] arguments)
        {       Cube cube = new Cube();
                cube.display();
                cube.area();
                cube.volume();
}       }
```

Hierarchical Inheritance:

when a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then this **type of inheritance** is known as **hierarchical inheritance**.



Hierarchical Inheritance

```
class A
{ public void methodA()
{ System.out.println("method of Class A"); }
}
class B extends A
{ public void methodB()
{ System.out.println("method of Class B"); }
}
class C extends A
{ public void methodC()
{ System.out.println("method of Class C"); }
}
class D extends A
{ public void methodD()
{ System.out.println("method of Class D"); }
}
```

```
class JavaExample
{
   public static void main(String args[])
   { B obj1 = new B();
     C obj2 = new C();
     D obj3 = new D();
     obj1.methodA();
     obj2.methodA();
     obj3.methodA();
} }
```

Method Overriding:

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in java**.

In other words, If subclass provides the specific implementation of the method that has been provided by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

1. method must have same name as in the parent class
2. method must have same parameter as in the parent class.
3. must be IS-A relationship (inheritance).

```
class Vehicle
{
        void run()
        {
                System.out.println("Vehicle is running");
        }
}
class Bike extends Vehicle
{
        void run()
        {
                System.out.println("Bike is running safely");
        }
        public static void main(String args[])
        {
                Bike obj = new Bike();
                obj.run();
        }
}
```

FINAL CLASS: (Inhibiting Inheritance of Class Using Final)

When a class is declared with the **final** keyword, it is called a **final class**. A final class cannot be **extended(inherited)**.

There are two uses of a final class:

1: One is to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer, Float, etc. are final classes. We can not extend them.

2: The other use of final with classes is to create an immutable class(once an object is created, we cannot change its content) like the predefined String class. One can not make a class immutable without making it final.

Syntax:

```
final class A
{
        // methods and fields
}
// The following class is illegal – because final class cannot be inherited
class B extends A
{
        // COMPILE-ERROR! Can't subclass A
}
```

```
public final class FinalClass
{
        public void display(int num)
        {
            System.out.println(“Sup-Int argument:" + num);
        }
}
public class DerievedClass extends FinalClass // Compile Error – Finale class cannot be inherited
{
        public void display(int num)
        {
            System.out.println(“Sub-Int argument:" + num);
        }
}
```

APPLICATION OF KEYWORD SUPER:

- The **super keyword in Java** is a reference variable that is used to refer to parent class when we're working with objects.

Characteristics of Super Keyword in Java:

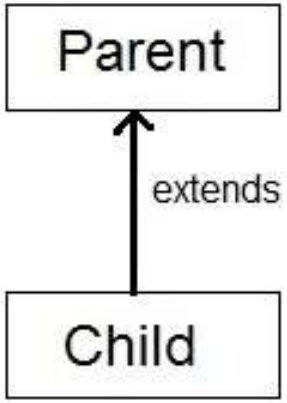
- **Super is used to call a superclass constructor:** When a subclass is created, its constructor must call the constructor of its parent class. This is done using the super() keyword, which calls the parent class's constructor.
- When calling a superclass constructor, the super() statement must be the first statement in the subclass's constructor.
- **Super is used to call a superclass method:** A subclass can call a method defined in its parent class using the super keyword. This is useful when the subclass wants to invoke the parent class's implementation of the method in addition to its own.
- **Super is used to access a superclass field:** A subclass can access a field defined in its parent class using the super keyword. This is useful when the subclass wants to reference the parent class's version of a field.
- **Super cannot be used in a static context:** The super keyword cannot be used in a static context, such as in a static method or a static variable initializer.

```
class UseOfSuper
{
   String EmpId;
   UseOfSuper()
   {
     EmpId="E19001";
     System.out.println("super class constructor called");
   }
   void display()
   {
      System.out.println("Employee Id is="+EmpId);
   }
}
```

```
class SubClass extends UseOfSuper
{   int salary;
    SubClass()
    {   super();
        salary=50000;
        System.out.println("sub class constructor called");
    }
    void display()
    {  super.display();
        System.out.println("Employee Id is="+super.EmpId);
        System.out.println("Salary is="+salary);
    }
    public static void main(String ar[])
    {  SubClass sb=new SubClass();
        sb.display();
    }
}
```

Runtime Polymorphism or Dynamic method dispatch

- Dynamic method dispatch is a mechanism by which a call to an overridden method is resolved at runtime, rather than compile time.
- This is how java implements runtime polymorphism. When an overridden method is called by a reference, java determines which version of that method to execute based on the type of object it refer to. In simple words the type of object which it referred determines which version of overridden method will be called.




```
Parent p = new Parent( );

Child c = new Child( );

Parent p = new Child( );
```

Upcasting

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

incompatible type

Upcasting:

When **Parent** class reference variable refers to **Child** class object, it is known as **Upcasting.**

```
class Game {
   public void type()
   {   System.out.println("Indoor & outdoor");
 }  }
class Cricket extends Game
{
      public void type()
      {   System.out.println("outdoor game");
      }
      public static void main(String[] args)
      {     Game gm = new Game();
            Cricket ck = new Cricket();
            gm.type();
            ck.type();
            gm=ck; //gm refers to Cricket object
            gm.type(); //calls Cricket's version of type
}     }
```

Difference between Static binding and Dynamic binding in java

Static Binding	Dynamic Binding
Static binding in Java occurs during compile time	Dynamic binding occurs during runtime
Static binding uses type(Class) information for binding	Dynamic binding uses instance of class(Object) to resolve calling of method at run-time
Overloaded methods are bonded using static binding	Overridden methods are bonded using dynamic binding at runtime
Static binding means when the type of object which is invoking the method is determined at compile time by the compiler	Dynamic binding means when the type of object which is invoking the method is determined at run time by the compiler

ABSTRACT : A class that is declared using "**abstract**" keyword is known as abstract class. It can have abstract methods (methods without body) as well as concrete methods (regular methods with body). An abstract class is never instantiated. It is used to provide abstraction.

Why we need an abstract class?

- Lets say we have a class Animal that has a method sound() and the subclasses of it like Dog, Lion, Horse, Cat etc. Since the animal sound differs from one animal to another, there is no point to implement this method in parent class.
- This is because every child class must override this method to give its own implementation details, like Lion class will say "Roar" in this method and Dog class will say "Woof".
- So when we know that all the derived classes will and should override the bass class method, then there is no point to implement this method in base class.
- Since the base class has an abstract method, you must need to declare this class as abstract.

When to use Abstract Methods & Abstract Class?

- Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Points to Remember

1. Abstract classes are not Interfaces.
2. If any class has even a single abstract method, then it must be declared as abstract.
3. Abstract classes can have Constructors, Member variables and Normal methods.
4. Abstract classes are never instantiated.
5. When you extend Abstract class with abstract method, you must define the abstract method in the child class.

```
abstract class Animal
{
	public abstract void sound(); //abstract method
}

public class Dog extends Animal
{
	public void sound()
	{
		System.out.println("Woof");
	}
	public static void main(String args[])
	{
		Dog obj = new Dog();
		obj.sound();
	}
}
```

Java Garbage Collection:

- In java, garbage means unreferenced objects.
- Garbage Collection is process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.
- To do so, we were using free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

Advantage of Garbage Collection:

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

INTERFACE:

•An **interface in java** is a blueprint of a class. It has static constants and abstract methods.

•The interface in java is **a mechanism to achieve 100% abstraction**. There can be only abstract methods in the java interface, no method body. It is used to implement multiple inheritance in Java.

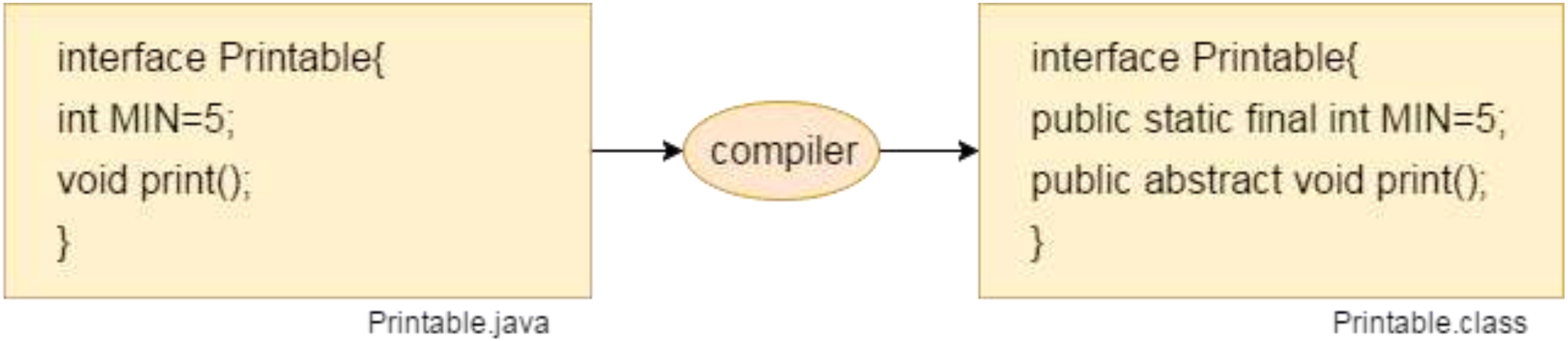
•Java allows a class to implement multiple interfaces, which can **declare methods but not provide implementations**, thus sidestepping the **ambiguity** by requiring the implementing class to provide its own concrete implementations.

•Java Interface also **represents IS-A relationship**.

•To declare an interface, use **interface** keyword. It is used to provide total abstraction. That means all the methods in interface are declared with empty body and are public and all fields are public, static and final by default.

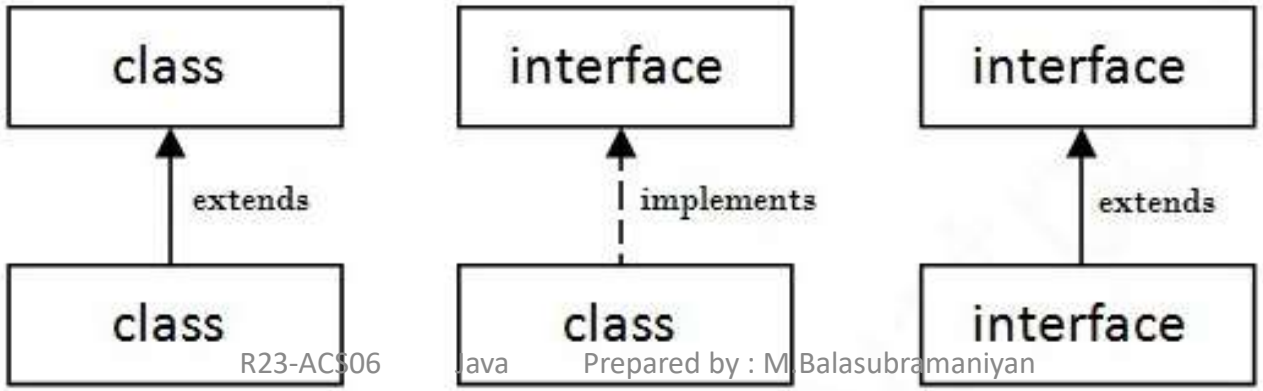
•A class that implement interface must implement all the methods declared in the interface. To implement interface use **implements** keyword.

In other words, Interface fields are public, static and final by default, and methods are public and abstract.



Relationship between classes and interfaces:

As shown in the figure given below, a class extends another class, an interface extends another interface but a **class implements an interface**.



- It is also used to achieve loose coupling.
- We can't create instance(interface can't be instantiated) of interface but we can make reference of it that refers to the Object of its implementing class.
- A class can implement more than one interface.
- An interface can extends another interface (but only one interface).

Syntax :

```
interface <interface_name>
{ // declare constant fields
// declare methods that abstract
// by default. }
```

Example:

```
// A simple interface
interface Player
{    final int id = 10;
     int move();
}
```

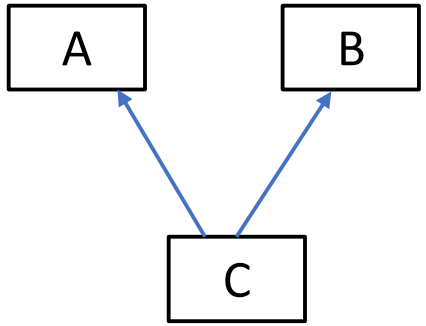
```
interface Drawable
{
        int x=5;
        void draw();
}
// Interface Implementation
class Rectangle implements Drawable
{
        public void draw()
        {
                x=10;
                System.out.println("drawing rectangle="+x);
        }
        public static void main(String args[])
        {
                Rectangle d=new Rectangle();
                d.draw();
        }
}
```

MULTIPLE INTERFACE:

```
interface Print
{
    void show();
}
```

```
interface Print1
{
    void show();
}
```

```
class A implements Print,Print1
{
        public void show()
        {
                System.out.println("Welcome");
        }
        public static void main(String args[])
        {
                A obj = new A();
                obj.show();
        }
}
```



NESTED INTERFACES IN JAVA:

- An interface defined inside another interface or class is known as nested interface. Nested interface is also known as inner interfaces or member interfaces. Nested interfaces are generally used to group related interfaces.

Key points about nested interfaces are :

- Nested interfaces are static by default, whether you declare them static or not.
- Nested interfaces are accessed using the outer interface or class name.
- Nested interfaces declared inside a class can have any access modifier, while nested interfaces declared inside another interface are public by default.
- Classes implementing inner interface are required to implement only inner interface methods, not outer interface methods.
- Classes implementing outer interface are required to implement only outer interface methods, not inner interface methods.

Use of nested interface in java:

- Nesting of interfaces is a way of logically grouping interfaces which are related or used at one place only.
- Nesting of interfaces helps to write more readable and maintainable code.
- It also increases encapsulation.

Syntax of declaring nested interface **using Interface**:

```
interface OuterInterfaceName
{
  interface InnerInterfaceName
  {
        // constant declarations
        // Method declarations
   }
 }
```

Syntax of declaring nested interface **using Class**:

Example :

```
class MyInterface
{
  interface MyInnerInterface
  {
        // constant declarations
        // Method declarations
   }
 }
```

JAVA PROGRAM OF NESTED INTERFACE DECLARED INSIDE AN INTERFACE:

```
interface MyInterface
{
   void calculateArea();
   interface MyInnerInterface
   {
      int id = 20;
      void print();
    }
 }
```

```
class NestedInterface implements MyInterface.MyInnerInterface
{
   public void print()
   {
     System.out.println("Print method of nested interface");
    }
   public static void main(String args [])
   {
     NestedInterface obj = new NestedInterface();
     obj.print();
     System.out.println(obj.id);
   }
}
```

JAVA PROGRAM OF NESTED INTERFACE DECLARED INSIDE A CLASS:

```
class OuterClass
{
   interface MyInnerInterface
   {
      int id = 20;
      void print();
   }
 }
```

```
class NestedInterfaceDemo implements OuterClass.MyInnerInterface
{
   public void print()
   {
     System.out.println("Print method of nested interface");
    }
   public static void main(String args [])
   {
     NestedInterfaceDemo obj = new NestedInterfaceDemo();
     obj.print();
     System.out.println(obj.id);
    }
}
```

DEFAULT METHODS IN INTERFACES:

- Before Java 8, interfaces could have only abstract methods.
- The implementation of these methods has to be provided in a separate class.
- So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface.
- To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface.
- The default methods were introduced to provide backward compatibility so that existing interfaces can use the lambda expressions without implementing the methods in the implementation class.
- Default methods are also known as **defender methods** or **virtual extension methods**.

```java
// methods in java
interface TestInterface
{
    // abstract method
    public void square(int a);

    // default method
    default void show()
    {
      System.out.println("Default Method Executed");
    }
}
```

```java
class TestClass implements TestInterface
{
    // implementation of square abstract method
    public void square(int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // default method executed
        d.show();
    }
}
```

STATIC METHODS IN INTERFACE:

- **Static Methods** in **Interface** are those methods, which are defined in the interface with the keyword static.
- Unlike other methods in Interface, these static methods contain the complete definition of the function and since the definition is complete and the method is static, therefore these methods cannot be overridden or changed in the implementation class.
- Similar to Default Method in Interface, the static method in an interface can be defined in the interface, but cannot be overridden in Implementation Classes.
- To use a static method, the Interface name should be instantiated with it, as it is a part of the Interface only.

Syntax:

```
interface InterfaceName
{
   <static> <ReturnType><MethodName>()
   {
      //Statements;
   }
   // Public and Abstract Method of Interface
   <returntype><methodname>(Parameter);
}
```

STATIC METHODS IN INTERFACE:

```
// A simple Java program to TestClassnstrate static
// methods in java
interface TestInterface
{
    // abstract method
    public void square (int a);
    // static method
    static void show()
    {
       System.out.println("Static Method Executed");
    }
}
```

```
class TestClass implements TestInterface
{
        // Implementation of square abstract method
        public void square (int a)
        {
                System.out.println(a*a);
        }
        public static void main(String args[])
        {
                TestClass d = new TestClass();
                d.square(4);

                // Static method executed
                TestInterface.show();
        }
}
```

PACKAGE:

- A set of **classes and interfaces** grouped are known as Packages in JAVA. The name itself defines that pack (group) of related types such as classes, sub-packages, enumeration, annotations, and interfaces that provide name-space management. Every class is a part of a certain package. When you need to use an existing class, you need to add the package within the Java program.

The benefits of using Packages in Java are as follows:

- The packages organize the group of classes into a single API unit.
- It will control the naming conflicts.
- The access protection will be easier. Protected and default are the access level controls to the package.
- Easy to locate the related classes.
- Reuse the existing classes in packages.

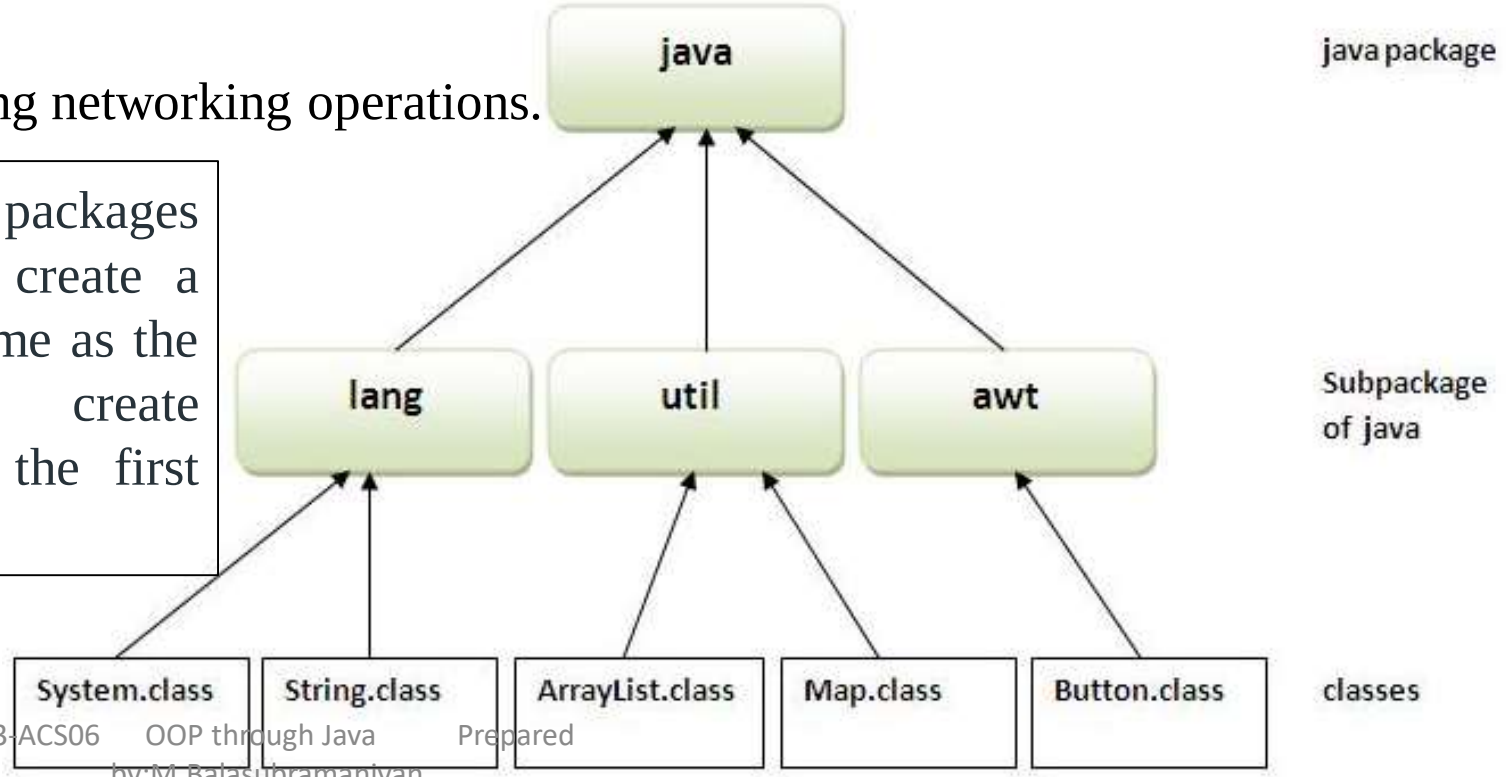
Types of packages:

- Built-in Packages
- User-defined packages

Built-in Packages:

1. **java.lang:** Contains language support classes(e.g classes that define primitive data types, math operations). This package is automatically imported.
2. **java.io:** Contains classes for supporting input/output operations.
3. **java.util:** Contains utility classes that implement data structures like Linked List, Dictionary, and support; for Date / Time operations.
4. **java.applet:** Contains classes for creating Applets.
5. **java.awt:** Contain classes for implementing the components for graphical user interfaces (like buttons, menus etc).
6. **java.net:** Contains classes for supporting networking operations.

User-defined packages: These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names.**



Simple example of java package

- The **package keyword** is used to create a package in java.

```
package mypack;
public class Simple
{  public static void main(String args[])
   {    System.out.println("Welcome to package");     }
}
```

Steps to compile java package:

you need to follow the **syntax** given below:

1.javac -d directory javafilename

For **example**

1.javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

Java.lang package in Java:

Provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Classes in Java.lang package :

- **Boolean**: The Boolean class wraps a value of the primitive type boolean in an object.
- **Byte**: The Byte class wraps a value of primitive type byte in an object.
- **Character.Subset**: Instances of this class represent particular subsets of the Unicode character set.
- **Double**: The Double class wraps a value of the primitive type double in an object.
- **Enum**: This is the common base class of all Java language enumeration types.
- **Float:** The Float class wraps a value of primitive type float in an object.
- **Integer**: The Integer class wraps a value of the primitive type int in an object.
- **ClassLoader**: A class loader is an object that is responsible for loading classes.

Java.util classes:

A utility class in Java is a class that provides common utility methods that are not tied to a specific object or instance. These classes often contain static methods meaning you can call them directly on the class without creating an instance of the class.

Here are some frequently used utility classes in Java and examples

- **java.util.Arrays**
- **java.util.Collections**
- **java.util.Date**
- **java.util.StringTokenizer**
- **java.util.Random**
- **java.util.regex.Pattern**
- **java.util.Scanner**
- **java.util.Calendar**

Steps to run java package program:

You need to use fully qualified name. for e.g. mypack.Simple etc to run the class.

To Run: java mypack.Simple
Output: Welcome to package

Accessing package from another package:

There are three ways to access the package from outside the package.

1.import package.*;
2.import package.classname;
3.fully qualified name.

EX: pack.A obj = **new** pack.A();//using fully qualified name

1) Using packagename.*

- If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages.
- The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg()
    {     System.out.println("Hello");
} }
//save by B.java
package mypack;
import pack.*;
 class B{
    public static void main(String args[])
    {   A obj = new A();
        obj.msg();
} }
```

2.Using packagename.classname:

- If you import package.classname then only declared class of this package will be accessible.

Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");  }
}
public class C{
  public void display(){System.out.println("Java Programming");  }
}
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
   obj.display();
  }
}
```

3) Using a fully qualified name:

If you use a fully qualified name, only the declared class of this package will be accessible. - Now, there is no need to import. But you need to use a fully qualified name every time you access the class or interface.

- It is generally used when two packages have the same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){ System.out.println("Hello"); }
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

```
//Save by A1.java
package pack1;
public class A
{
     public void msg()
     {
          System.out.println("Hai");
      }
}
```

MEMBER ACCESS RULES:

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are –

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword

- Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.
- A variable or method declared without any access control modifier is visible to any other class in the same package. The fields in an **interface** are implicitly **public static final** and the **methods** in an interface are by **default public**.
- In the following example, we have created two packages: my pack and class A. We are accessing class **A** from outside its package since class **A** is not public and cannot be accessed from outside the package.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}

//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

Private Access Modifier – Private:

- Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.
- Private access modifier is the most restrictive access level. **Class** and **interfaces** cannot be private.
- Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.
- Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example:

The following class uses private access control –

```
class A
{       private int data=40;
        private void msg()
        {
                System.out.println("Hello java");
}       }
public class Simple
{       public static void main(String args[])
        {       A obj=new A();
                System.out.println(obj.data);//Compile Time Error
                obj.msg();//Compile Time Error
}       }
```

In the above example, we have created two classes **A** and **Simple**. **A** class contains private data member and private method. We are accessing these private members from outside the class, so there is compile time error.

Public Access Modifier – Public:

- A class, field, method, constructor, interface. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

- However, if the **public class** we are trying to access is **in a** different **package**, then the public class still **needs to be imported**. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

```
//save by A.java
package pack;
public class A
{
    public void msg()
    {   System.out.println("Hello");
    }
}
//save by B.java
package mypack;
import pack.*;

class B
{
  public static void main(String args[])
  {
        A obj = new A();
        obj.msg();
  }
}
```

Output : Hello

Protected Access Modifier – Protected:

- Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in another package or any subclasses within the package of the protected members' class.
- The protected access modifier cannot be applied to **classes** and **interfaces.** Methods, and fields can be declared protected in a class, however, **methods** and **fields** in an interface cannot be declared protected.
- The protected access modifier is accessible within the package and **outside the package but through inheritance only**.

Example:

```
//save by A.java
package pack;
public class A
{
  protected void msg()
  {
     System.out.println("Hello");
  }
}
```

```
//save by B.java
package mypack;
import pack.*;
class B extends A
{
  public static void main(String args[])
  {
    B obj = new B();
    obj.msg();
  }
}
```

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

Understanding all java access modifiers

Let's understand the access modifiers by a simple table.

Access Modifier	Within Class	Within Package	Outside Package by Subclass only	Outside Package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Wrapper Classes:

A Wrapper class in Java is a class whose object wraps or contains primitive data types. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object. Let's check on the wrapper classes in Java with examples:

Need of Wrapper Classes

There are certain needs for using the Wrapper class in Java as mentioned below:

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Advantages of Wrapper Classes

1. Collections allowed only object data.
2. On object data we can call multiple methods compareTo(), equals(), toString()
3. Cloning process only objects
4. Object data allowed null values.
5. Serialization can allow only object data.

Autoboxing:

The automatic conversion of primitive types to the object of their corresponding wrapper classes is known as autoboxing. For example – conversion of int to Integer, long to Long, double to Double, etc.

```
import java.util.ArrayList;
class Autoboxing {
    public static void main(String[] args)
    {
        char ch = 'a';
        // Autoboxing- primitive to Character object
        // conversion
        Character a = ch;
        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        // Autoboxing because ArrayList stores only objects
        arrayList.add(25);
        // printing the values from object
        System.out.println(arrayList.get(0));
    }
}
```

Output:
25

Unboxing:

It is just the reverse process of autoboxing. Automatically converting an object of a wrapper class to its corresponding primitive type is known as unboxing. For example – conversion of Integer to int, Long to long, Double to double, etc.

```
import java.util.ArrayList;

class Unboxing {
    public static void main(String[] args)
    {
        Character ch = 'a';
        // unboxing - Character object to primitive conversion
        char a = ch;
        ArrayList<Integer> arrayList = new ArrayList<Integer>();
        arrayList.add(24);
        // unboxing because get method returns an Integer object
        int num = arrayList.get(0);
        // printing the values from primitive data types
        System.out.println(num);
    }
}
```

Output:
24

Java Date and Time:

The java.time, java.util, java.sql and java.text packages contain classes for representing date and time. The following classes are important for dealing with date in Java.

- java.time.LocalDate class
- java.time.LocalTime class
- java.time.LocalDateTime class
- java.time.MonthDay class
- java.time.OffsetTime class
- java.time.OffsetDateTime class
- java.time.Clock class
- java.time.ZonedDateTime class
- java.time.ZoneId class
- java.time.ZoneOffset class
- java.time.Year class
- java.time.YearMonth class
- java.time.Period class
- java.time.Duration class
- java.time.Instant class
- java.time.DayOfWeek enum
- java.time.Month enum

Java Date and Time APIs

- Java provide the date and time functionality with the help of two packages java.time and java.util. The package java.time is introduced in Java 8, and the newly introduced classes tries to overcome the shortcomings of the legacy java.util.Date and java.util.Calendar classes.
- Classical Date Time API Classes
- The primary classes before Java 8 release were:
- **Java.lang.System:** The class provides the currentTimeMillis() method that returns the current time in milliseconds. It shows the current date and time in milliseconds from January 1st 1970.
- **java.util.Date:** It is used to show specific instant of time, with unit of millisecond.
- **java.util.Calendar:** It is an abstract class that provides methods for converting between instances and manipulating the calendar fields in different ways.
- **java.text.SimpleDateFormat:** It is a class that is used to format and parse

Display Current Date:

To display the current date, import the java.time.LocalDate class, and use its now() method:

```
import java.time.LocalDate; // import the LocalDate
class public class Main
{
        public static void main(String[] args)
        {
                LocalDate myObj = LocalDate.now(); // Create a date object
                System.out.println(myObj); // Display the current date
        }
}
```

Output:

        2024-11-20

ENUMERATION:

- An enumeration (enum for short) in Java is a special data type which contains a set of predefined constants.
- You'll usually use an enum when dealing with values that aren't required to change, like days of the week, seasons of the year, colors, and so on.

Creation of an Enum in Java:

- To create an enum, we use the enum keyword, similar to how you'd create a class using the class keyword.
- **Syntax:**
- enum <enumname>

```
{
        value 1,
        Value 2      }
```

- **Example:**

```
enum Colors
{
        RED,
        BLUE,
        YELLOW,
        GREEN
 }
```

- You will not get an error if the values are assigned in lowercase.
- Each value in an enum is separated by a comma.

We can create a new variable and assign one of the values of our enum to it as follows,

```
enum Colors
{
        RED,
        BLUE,
        YELLOW,
        GREEN
}
public class Main
{
        public static void main(String[] args)
        {
                Colors red = Colors.RED;
                System.out.println(red);
        }
}
```

Output: RED

This is similar to **initializing any other variable**. In the code above, we initialized a Colors variable and assigned one of the values of an enum to it using the dot.

syntax: Colors red = Colors.RED;.

Note: We can create our **enum inside the Main class** and the code will still work.

MATH CLASS:

- The **Java Math class**, part of the **java.lang** package is a utility class that provides a wide range of mathematical functions and constants. This class is essential for performing basic mathematical operations and is crucial in many Java applications, particularly those involving scientific and engineering calculations.
- The **methods in the Math class are all static**, so you can call them directly on the class without needing to create an instance of it.

Declaration:

```
public final class Math extends Object
{
        // Class body with methods and constants
}
```

This declaration indicates that the Math class:

•**Cannot be subclassed:** Because it's declared as final.

•**Inherits from Object:** Like all Java classes, Math implicitly extends the Object class, which is the root of the Java class hierarchy.

Example:

One of the most commonly used methods in Math class is **sqrt()**, which calculates the square root of a number.

```
public class MathExample
{
     public static void main(String[] args)
    {
        double value = 9;
        double squareRoot = Math.sqrt(value);
        System.out.println("The square root of " + value + " is " + squareRoot);
 }  }
```

Output: *The square root of 9.0 is 3.0*

•Java Math class provides functions like **exp()**, **log()**, **sqrt()**, and trigonometric functions (**sin()**, **cos()**, **tan()**, etc.), which are essential in **scientific calculations**.

•Functions for power and logarithmic calculations are useful in engineering domains like **signal processing, strength of materials, and electrical engineering**.

•Methods like **ceil()**, **floor()**, and **round()** are used in **financial applications** for rounding off the values to the nearest integer, which is crucial in monetary calculations.

•Functions like **max()**, **min()**, and **random()** are useful for **statistical analysis** and generating random data for simulations or modeling.

•Although more advanced libraries are often used for **complex calculations**, the Math class still provides foundational mathematical functions essential in preprocessing and transforming data.

EXCEPTION HANDLING:

The **exception handling in java** is one of the powerful *mechanism to handle the runtime errors* so that normal flow of the application can be maintained.

What is exception:

In java, exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

What is exception handling:

Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

Advantage of Exception Handling:

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
```

NOTE: Suppose there is 7 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 and 7 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

TYPES OF EXCEPTIONS:

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1.Checked Exception
2.Unchecked Exception
3.Error

1) Checked Exception:

- Checked exceptions, also known as compile-time exceptions, are exceptions that must be either caught or declared in the method signature using the throws keyword.
- These exceptions are typically used to handle expected error scenarios that a program can recover from. They force the developer to acknowledge and handle these exceptional conditions, ensuring that appropriate actions are taken.
- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions.

- Some common examples of checked exceptions in Java include:

- e.g. IOException, SQLException, FileNotFoundException, etc. Checked exceptions are **checked at compile-time**.

Uncaught Exceptions:

- Uncaught Exceptions explains how the exception is going to be handled if the exception is not handled by our own, then it shows how the exceptions is thrown by default by java.

- We take the following example which as runtime exception.

```
class exp
{
	public static void main(String arg[])
	{
		int d=0;
		int a=42/d;
	}
}
```

The above program causes a divide-by-zero error. Whenever if there is an exception thrown, then it check for the exception handlers of our own.

- If there is no exception handler by the user, then immediately the **exception is caught by the default handler provided by the Java run-time system**.
- The default handler displays a string describing the exception, **prints a stack trace** from the point at which the **exception occurred and terminates the program**.
- Here is the exception generated when the previous example is executed.

exception in thread "main" java.lang.ArithmeticException: / by zero

at exp.main(exp.java:6)

Note: From the exception string message note that how the class name-**exp**, the method name-**main**, the file name-**exp.java**, the line number **6** and the type of exception thrown is a subclass of **Exception** called **ArithmeticException** are all included in the simple stack trace.

2) Unchecked Exception:

- Unchecked exceptions, also known as runtime exceptions, are exceptions that do not need to be caught explicitly or declared using the throws keyword.

- They usually represent programming errors, such as invalid calculations or incorrect usage of APIs, that can be prevented by proper code design and testing.

- The classes that extend RuntimeException are known as unchecked exceptions. Unchecked exceptions are not checked at compile-time rather they are **checked at runtime**.

- unchecked exceptions typically indicate issues that should be fixed during development, they can often be avoided through proper coding practices, such as input validation and defensive programming.

Some common exceptions (unchecked Exception):

ArithmeticException:

- If we divide any number by zero, there occurs an ArithmeticException.

```
Ex: int a=50/0;//ArithmeticException
```

NullPointerException:

- If we have null value in string reference variable, performing any operation by the variable occurs an NullPointerException.

```
Ex: String s=null;
    System.out.println(s.length());//NullPointerException
```

NumberFormatException:

- The wrong formatting of any value, may occur NumberFormatException. Suppose you have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
Ex: String s="abc";
    int i=Integer.parseInt(s);//NumberFormatException
```

ArrayIndexOutOfBoundsException:

- If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
Ex: int a[]=new int[5];
    a[10]=50; //ArrayIndexOutOfBoundsException
```

Exception Handling Keywords:

There are 5 keywords used in java for exception handling.

1.try

2.catch

3.throw

4.throws

5. finally.

Usage of try, catch, throw:

try block:

Java try block is used to enclose the code that might throw an exception. The code which the programmer want to monitor can be added within the try block. It must be used within the method.

Java try block must be followed by either catch or finally block.

Syntax of java try-catch

```
try{
//code that may throw exception
}
catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw exception
}
finally{}
```

catch block:

- Catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.
- You can use multiple catch block to handle individual exception for a single try.

Program without exception handling:

Let's try to understand the problem if we don't use try-catch block.

```
public class Testtrycatch1
{    public static void main(String args[])
     {
          int data=50/0;//may throw exception
          System.out.println("rest of the code...");
}    }
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

As displayed in the above example, **rest of the code** output is not executed.

Program with exception handling:

Let's see the solution of above problem by java try-catch block.

```
public class Testtrycatch2{
  public static void main(String args[]){
   try{
      int data=50/0;
         }
catch(ArithmeticException e)
          {   System.out.println(e);   }
   System.out.println("rest of the code...");
}
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

Multi catch block:

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

simple example of java multi-catch block.

```
public class tmcb
{
   public static void main(String args[])
   {
         try
         {
                  int a[]=new int[5];
                  a[5]=30/5;
                  System.out.println(a[5]);
         }
catch(ArithmeticException e)
{
   System.out.println("Arithmetic error");
}

catch(ArrayIndexOutOfBoundsException e)
{
   System.out.println("Array index out of bounds");
}

catch(Exception e)
{
   System.out.println("common task completed");
}
System.out.println("rest of the code");
}
}
```

Throw:

The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exceptions. The throw keyword is mainly used to throw custom exceptions.

Syntax:

`throw` *`Instance`*

Example:

```
throw new ArithmeticException("/ by zero");
```

Example:

```
public class TestThrow1 {
  static void validate(int age) {
    if(age<18)
        throw new ArithmeticException(“you are not eligible");
    else
        System.out.println("welcome to vote");
  }
  public static void main(String args[]){
     TestThrow1.validate(13);
     System.out.println("rest of the code..."); }
  }
```

Throws keyword:

Throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

Syntax of java throws

```
return_type method_name() throws exception_list
{
        //method code
}
```

- In a program, if there is a chance of rising an exception then compiler always warn us about it and compulsorily we should handle that checked exception, Otherwise we will get compile time error.
- To prevent this compile time error we can handle the exception in two ways:

1. try-catch
2. **throws** keyword

- We can use throws keyword to delegate the responsibility of exception handling to the caller (It may be a method or JVM), and then the caller method is responsible for handling that exception.

Example program for throws keyword:

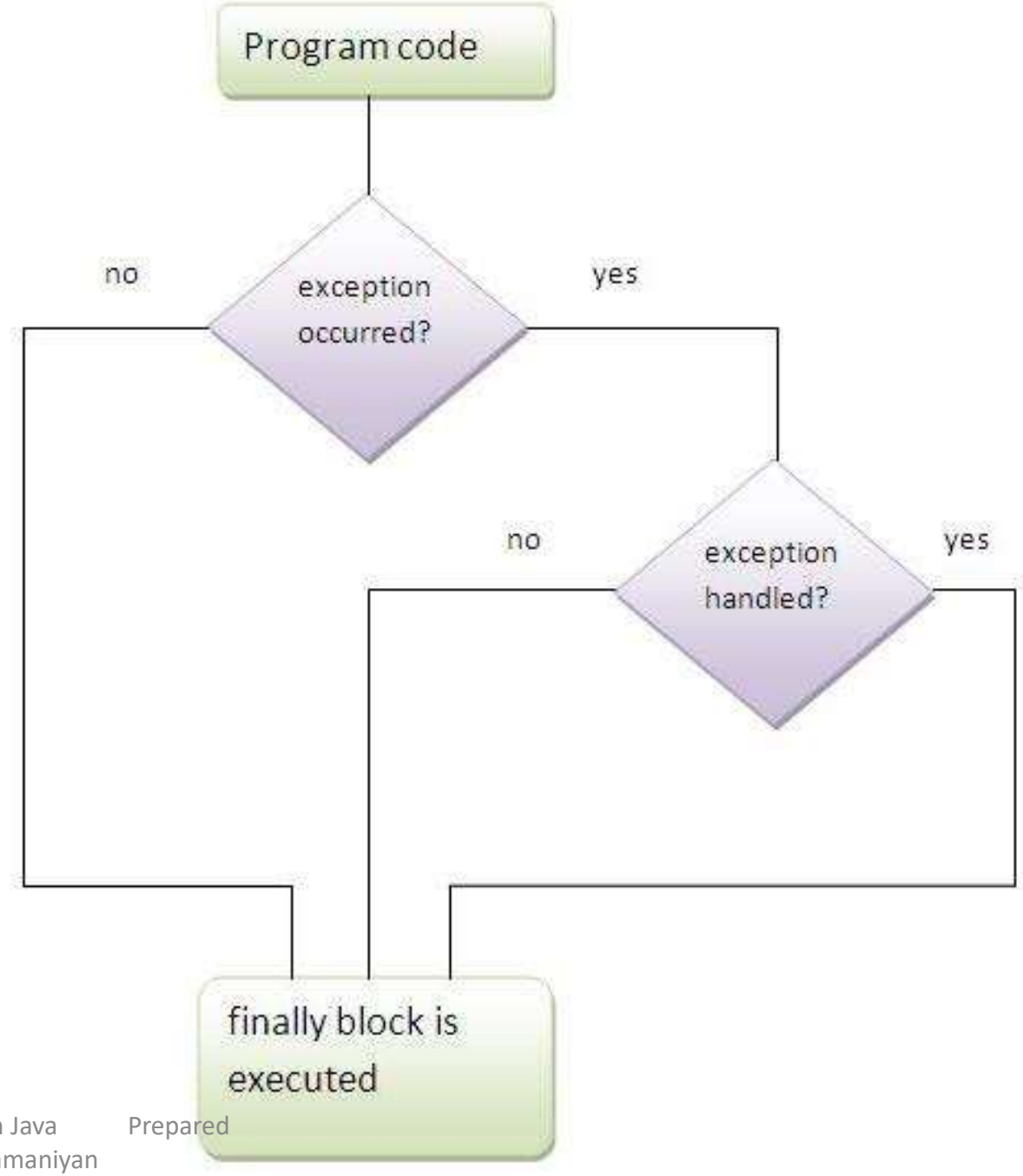
- It provides information to the caller of the method about the exception.

```
class ThrowsExecp
{
    static void fun() throws IllegalAccessException
    {
        System.out.println("Inside fun(). ");
        throw new IllegalAccessException();
    }
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalAccessException e)
        {
            System.out.println("caught in main.");
        }
    }
}
```

- Finally:

Finally block is a block that is used *to execute important code* such as closing connection, stream etc.

Java finally block is always executed whether exception is handled or not. Java finally block follows try or catch block.



Example: Exception occurs and not handled.

```
class TestFinally
{
  public static void main(String args[])
 {
  try
  {       int data=25/0;
          System.out.println(data);
  }
  catch(NullPointerException e)
  {
          System.out.println(e);
  }
  finally
  {
    System.out.println("finally block is
                              always executed");
  }
  System.out.println("rest of the code...");
  }
}
```

Output: finally block is always executed
exception in thread "main" java.lang.ArithmeticException: /
by zero at TestFinally.main(TestFinally.java:7)

Example: exception occurs and handled.

```
public class TestFinally
{
  public static void main(String args[])
  {
  try
  {       int data=25/0;
          System.out.println(data);
  }
  catch(ArithmeticException e)
  {
          System.out.println(e);
  }
  finally
  {
       System.out.println("finally block is always
                                  executed");
  }
  System.out.println("rest of the code...");
  }
}
```

Output: java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code....

Difference between throw and throws:

No.	throw	throws
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

Java throw example:-

```
void m()
{
  throw new ArithmeticException("sorry");
}
```

Java throws example:-

```
void m()throws ArithmeticException
{
        //method code
}
```

Difference between final, finally and finalize:

No.	final	finally	finalize
1)	Final is used to apply restrictions on class, method and variable. Final class can't be inherited, final method can't be overridden and final variable value can't be changed.	Finally is used to place important code, it will be executed whether exception is handled or not.	Finalize is used to perform clean up processing just before object is garbage collected.
2)	Final is a keyword.	Finally is a block.	Finalize is a method.

Throwable Class:

- The **Java Throwable** class is the superclass of all errors and exceptions in the Java language. Only objects that are instances of this class (or one of its subclasses) are thrown by the Java Virtual Machine or can be thrown by the Java throw statement.

Class Declaration:

Following is the declaration for **java.lang.Throwable** class –

- public class Throwable extends Object implements Serializable

Class constructors:

Sr.No.	Constructor & Description
1	**Throwable()** This constructs a new throwable with null as its detail message.
2	**Throwable(String message)** This constructs a new throwable with the specified detail message.
3	**Throwable(String message, Throwable cause)** This constructs a new throwable with the specified detail message and cause.
4	**Throwable(Throwable cause)** This constructs a new throwable with the specified cause and a detail message of (cause==null ? null : cause.toString()) (which typically contains the class and detail message of cause)

Example: Printing Stacktrace of Throwable

- The following example shows the usage of Java Throwable printStackTrace() method. We've defined a method raiseException() which throws a Throwable after setting the Stacktrace. In main method, raiseException() method is called and in catch block exception stack trace is retrieved and printed using printStackTrace() method.

```
package com.tutorialspoint;
public class ThrowableDemo
{
   public static void main(String[] args)
   {
      try
      {
         raiseException();
      }
      catch(Throwable e)
      {
    // prints stacktrace for this Throwable Object
          e.printStackTrace();
      }
   }
```

```
public static void raiseException() throws Throwable
{
Throwable t = new Throwable("This is new Exception...");
StackTraceElement[] trace = new StackTraceElement[]
{
new
StackTraceElement("ClassName","methodName","fileName",5)
}; // sets the stack trace elements t.setStackTrace(trace); throw t;
}
}
```

Java I/O | I/O Streams in Java:

- The **java.io** package is used to handle **input** and **output** operations. Java IO has various classes that handle input and output sources. A **stream** is a sequence of data.
- Java input stream classes can be used to read data from input sources such as **keyboard** or a **file**. Similarly output stream classes can be used to write data on a **display** or a **file** again.
- **A Stream is also a sequence of data.** It is neither a data structure nor a store of data. For example, a river stream is where water flows from source to destination. Similarly, these are data streams; data flows through one point to another.
- IO streams in Java help to **read** the data from an **input stream**, such as a file and **write** the data into an **output stream**, such as the standard display or a file again. It represents the **source as input** and the **destination as output**. It can handle all types of data, from primitive values to advanced objects.
- The **java.io** package consists of output and input streams used to write and read data to files or other output and input sources.

There are 3 categories of classes in java.io package:

- Input Streams, Output Streams, Error Streams.

Java supports three streams that are automatically attached with the console.

1. **System.out:** Standard output stream,
2. **System.in:** Standard input stream,
3. **System.err:** Standard error stream

Input Streams

- Input Streams help us read data from the input source. They are an abstract class that provides a programming interface for all input streams.
- Input streams are opened implicitly as soon as they are created. We use a close() method on the source object to close the input stream.

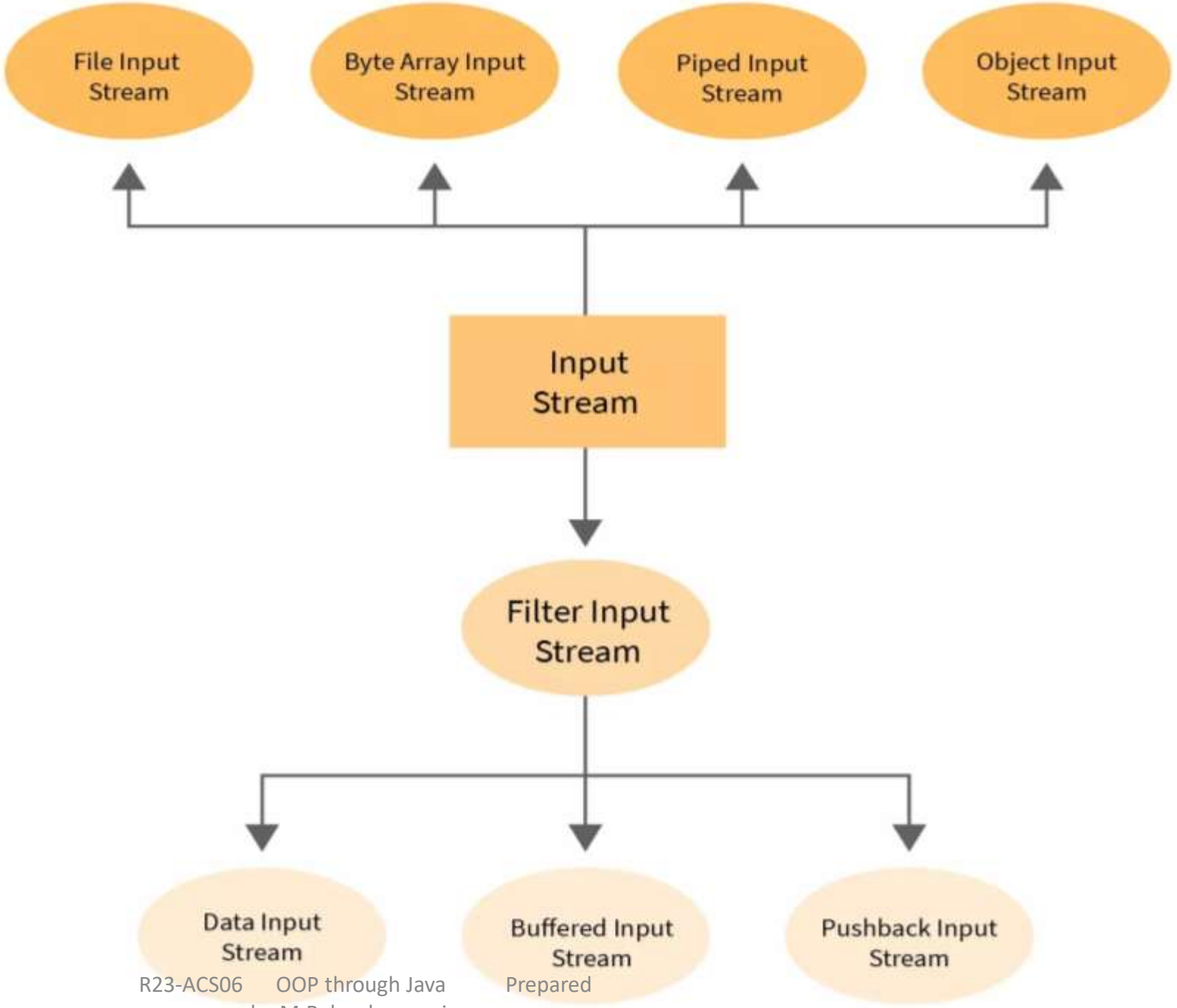
Output Streams

- The executed program's output must be stored in a file for further use. Output streams help us write data to an output source(such as a file). Similarly to input streams, output streams are abstract classes that provide a programming interface for all output streams.
- The output stream is opened as soon as it is created and explicitly closed using the "close() "method.

Error Streams

- Error streams are the same as output streams. In some ide's, the error is displayed in different colors (other than the color of the output color). It gives output on the console the same as output streams.

InputStream Hierarchy:

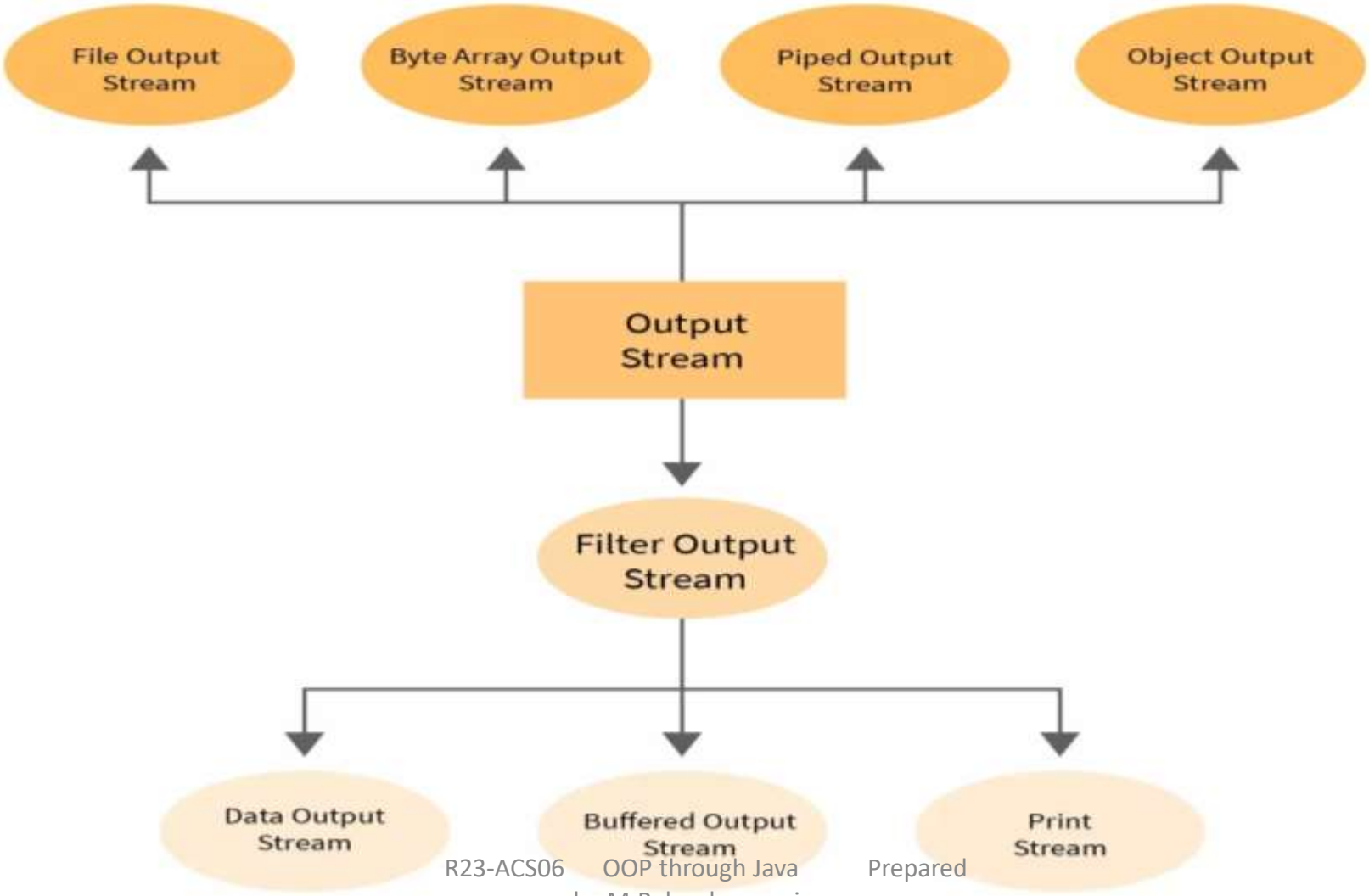


Methods of InputStream:

- **public abstract int read() throws IOException**
  - returns the data of the next byte in the input stream.
- **public int available() throws IOException**
  - **returns the number of bytes that can be read from the input stream.**
- **public void close() throws IOException**
  - **closes the current input stream and releases any associated system resources.**
- **public void mark(int readlimit)**
  - **It marks the current position in the input stream.**
- **public boolean markSupported()**
  - **It tells whether a particular input stream supports the mark() and reset() method.**
- **public int read(byte[ ] b) throws IOException**
  - **reads the bytes from the input stream and stores every byte in the buffer array.**
- **public int read(byte[ ] b , int off , len) throws IOException**
  - **It reads up to len bytes of data from the input stream and returns the total number of bytes stored in the buffer.**
- **public void reset() throws IOException**
  - **It repositions the stream to the last called mark position.**
- **public long skip(long n) throws IOException**
  - **This method discards n bytes of data from the input stream.**

Output Stream:

It is an abstract superclass of the java.io package that writes data to an output resource, which is, in other words, writing the data into files. We can create an object of the output stream class using the new keyword. The output stream class has several types of constructors.



Methods of OutputStream:

public void close() throws IOException

- This method closes the current output stream and releases any associated system resources.

public void flush() throws IOException

- It flushes the current output stream and forces any buffered output to be written out.

Public void write(byte[ ] b) throws IOException

- This method writes the b.length bytes from the specified byte array to the output stream.

Public void write(byte[ ] b ,int off ,int len) throws IOException

- It writes up to len bytes of data for the output stream.

Public abstract void write(int b) throws IOException

- The method above writes the specific bytes to the output stream. It does not return a value.

```
import java.io.*;
public class FileIO
{
  public static void main(String args[]) throws IOException
  {
      FileReader in = null;
      FileWriter out = null;
      try
      {
        in = new FileReader("D:\\Sw\\jdk22\\bin\\Input-File.txt");
      out = new FileWriter("D:\\Sw\\jdk22\\bin\\Output-File.txt");
         int c;
         while ((c = in.read()) != -1)
         {
             out.write(c);
          }
       System.out.println("Reading and Writing in a file is done");
       }
```

```
      catch(Exception e)
      {
          System.out.println(e);
       }
       finally
       {
           if (in != null)
           {
               in.close();
           }
           if (out != null)
          {
               out.close();
           }
       }
    }
}
```

Output: Reading and Writing in a file is done

ByteStream Classes in Java

- ByteStream classes are used to **read bytes** from the input stream and **write bytes** to the output stream. In other words, we can say that ByteStream classes **read/write the data of 8-bits.**
- We can **store video, audio, characters, etc.,** by using ByteStream classes. These classes are part of the java.io package.
- The **ByteStream** classes are divided into two types of classes, i.e., **InputStream and OutputStream**. These classes are abstract and the super classes of all the Input/Output stream classes.

CharacterStream Classes in Java

- The **java.io package** provides CharacterStream classes to overcome the limitations of ByteStream classes, which can only handle the 8-bit bytes and is not compatible to work directly with the Unicode characters.
- CharacterStream classes are used to work with **16-bit** Unicode characters. They can perform operations on characters, char arrays and Strings.
- However, the CharacterStream classes are mainly used to read characters from the source and write them to the destination. For this purpose, the **CharacterStream** classes are divided into two types of classes, i.e., **Reader class** and **Writer class.**

Scanner Class in Java(Refer – Unit 1 topic "User Input to Programs")

- In Java, Scanner is a class in java.util package used for obtaining the input of the primitive types like int, double, etc. and strings.

- Using the Scanner class in Java is the easiest way to read input in a Java program, though not very efficient if you want an input method for scenarios where time is a constraint like in competitive programming.

Java File Class Methods:

- In Java, a **File** is an abstract data type. A named location used to store related information is known as a **File**. There are several **File Operations** like **creating a new File, getting information about File, writing into a File, reading from a File** and **deleting a File**.

No	Method	Return Type	Description
1.	canRead()	Boolean	The **canRead()** method is used to check whether we can read the data of the file or not.
2.	createNewFile()	Boolean	The **createNewFile()** method is used to create a new empty file.
3.	canWrite()	Boolean	The **canWrite()** method is used to check whether we can write the data into the file or not.
4.	exists()	Boolean	The **exists()** method is used to check whether the specified file is present or not.
5.	delete()	Boolean	The **delete()** method is used to delete a file.
6.	getName()	String	The **getName()** method is used to find the file name.
7.	getAbsolutePath()	String	The **getAbsolutePath()** method is used to get the absolute pathname of the file.
8.	length()	Long	The **length()** method is used to get the size of the file in bytes.
9.	list()	String[]	The **list()** method is used to get an array of the files available in the directory.
10.	mkdir()	Boolean	The **mkdir()** method is used for creating a new directory.

File Operations in Java:

We can perform the following operation on a file:

- Create a File
- Get File Information
- Write to a File
- Read from a File
- Delete a File

No.	Method()	Description
1	**createNewFile()**	Is used to create a new file. This method returns true when it successfully creates a new file and returns false when the file already exists.
2	**write()**	Is used to write the data into the specified file.
3	read()	Is used to read the data from the specified file.
4	delete()	Is used to delete the specified file.
Getting file Information		
5.1	getName()	Getting file name
5.2	getAbsolutePath()	Getting the absolute path of the file
5.3	canWrite()	Checking whether the file is writable or not
5.4	canRead()	Checking whether the file is readable or not
5.5	length()	Getting the length of the file in bytes

STRING HANDLING IN JAVA

In Java, a **String** is an object that represents a sequence of characters. Java provides a robust and flexible API for handling strings, allowing for various operations such as **concatenation**, **comparison**, and **manipulation**.

CharSequence Interface

CharSequence Interface is used for representing the sequence of Characters in Java. Classes that are implemented using the CharSequence interface are mentioned below and these provides much of functionality like substring, lastoccurence, first occurence, concatenate, toupper, tolower etc.

1.String

String is an immutable class which means a constant and cannot be changed once created and if wish to change , we need to create an new object and even the functionality it provides like toupper, tolower, etc

Syntax

String str= “Hello";

or

String str= new String(“Hello")

2.StringBuffer

StringBuffer is a peer class of **String**, it is mutable in nature and it is thread safe class , we can use it when we have multi threaded environment and shared object of string buffer i.e, used by mutiple thread.

Syntax: StringBuffer demoString = new StringBuffer(“Welcome to Java Programming");

3.StringBuilder

StringBuilder in Java represents an alternative to **String and StringBuffer** Class, as it creates a mutable sequence of characters and it is not thread safe. It is used **only within the thread**, so there is no extra overhead , so it is mainly used for single threaded program.

Syntax: StringBuilder demoString = new StringBuilder();
demoString.append(“Java");

Java String compareTo() Method:

String Extraction:

substring()

The substring() method returns a new string that is a substring of the original string. It takes one or two arguments: the start index and optionally the end index.

Syntax:

```
public String substring(int beginIndex) public String substring(int beginIndex, int endIndex)
```

Example:

```
public class SubstringExample
{
    public static void main(String[] args)
    {
        String str = "Hello, World!";
        String substr1 = str.substring(7);
        String substr2 = str.substring(0, 5);
        System.out.println("Substring from index 7: " + substr1);
        System.out.println("Substring from index 0 to 5: " + substr2);
    }
}
```

Output:

```
Substring from index 7: World!
Substring from index 0 to 5: Hello
```

String Modifying:

replace()

The replace() method replaces all occurrences of a specified character or substring with a new character or substring.

Syntax:

```
public String replace(char oldChar, char newChar)
public String replace(CharSequence target, CharSequence replacement)
```

Example:

```
public class ReplaceExample
{
        public static void main(String[] args)
        {
                String str = "Hello, World!";
                String result1 = str.replace('o', '0’);
                String result2 = str.replace("World", "Java");
                System.out.println("Replaced 'o' with '0': " + result1);
                System.out.println("Replaced 'World' with 'Java': " + result2);
}       }
```

Output:

```
Replaced 'o' with '0’:
Hell0, W0rld! Replaced 'World' with 'Java': Hello, Java!
```

Compare two Strings:

Comparing strings is the most common task in different scenarios such as **input validation** or **searching algorithms**. The most common method to compare two strings in Java is equals(). This method compares the content of two strings for equality.

```
public class CompareStrings
{
   public static void main(String[] args)
   {
      String s1 = "Hello";
      String s2 = "Geeks";
      String s3 = "Hello";
      // Comparing strings
      System.out.println(s1.equals(s2));
      System.out.println(s1.equals(s3));
   }
}
```

Output:

```
false
true
```

Searching String:

indexOf()

The indexOf() method returns the index within the string of the first occurrence of the specified substring or character. It returns -1 if the substring or character is not found.

Syntax:

```
public int indexOf(int ch)
public int indexOf(int ch, int fromIndex)
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
```

Example:

```
public class IndexOfExample
{
    public static void main(String[] args)
    {
    String str = "Hello, World!";
    int index1 = str.indexOf('o’);
    int index2 = str.indexOf('o', 5); int index3 =
str.indexOf("World");
    int index4 = str.indexOf("world"); ': -1
```

```
// Case-sensitive
System.out.println("Index of 'o': " + index1);
System.out.println("Index of 'o' from index 5: " + index2);
System.out.println("Index of 'World': " + index3);
System.out.println("Index of 'world': " + index4);
}
}
```

Output:

Index of 'o': 4 Index of 'o' from index 5: 8 Index of 'World': 7 Index of 'world

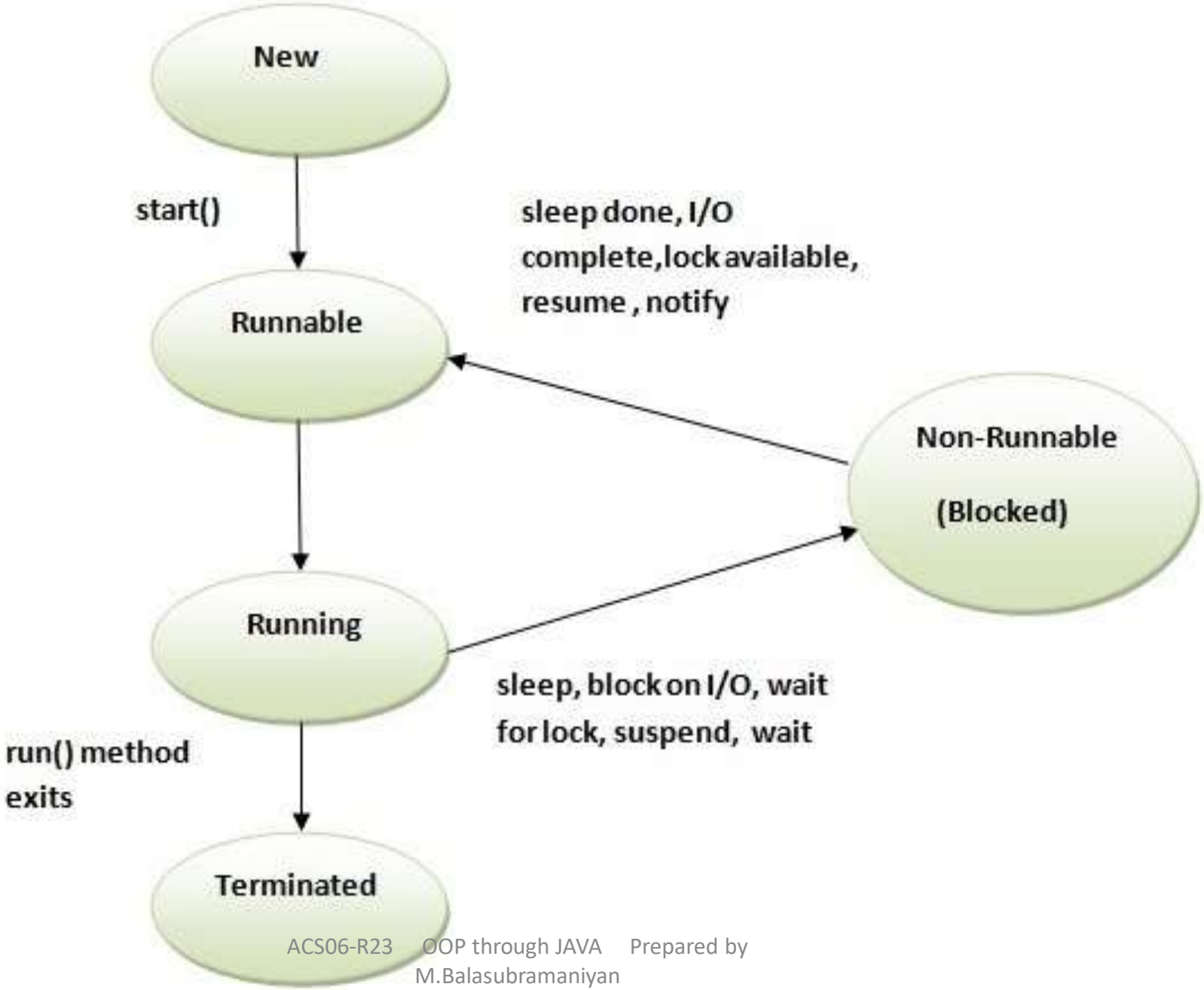
THREAD

A thread is a:

- Referred as a lightweight process.
- Facility to allow multiple activities within a single process.
- A thread is a series of executed statements.
- Each thread has its own program counter, stack, and local variables.
- A thread is a nested sequence of method calls.
- It shares memory, files, and per-process state.
- Every Java program creates at least one thread [main() thread]. Additional threads are created through the Thread constructor or by instantiating classes that extend the Thread class.

Life cycle of a Thread (Thread States):

- A thread can be in one of the five states. According to sun, there is only 4 states in **thread life cycle in java** new, runnable, non-runnable and terminated. There is no running state.
- But for better understanding the threads, we are explaining it in the 5 states.
- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:
    - **New**
    - **Runnable**
    - **Running**
    - **Non-Runnable (Blocked)**
    - **Terminated**


New
start()
Runnable
sleep done, I/O
complete,lock available,
resume , notify
Non-Runnable
(Blocked)
Running
sleep, block on I/O, wait
for lock, suspend, wait
run() method
exits
Terminated

Creating Threads:

Thread implementation in java can be achieved in two ways:

1.Extending the Thread class

2.Implementing the Runnable Interface

Note: The Thread and Runnable are available in the java.lang.* package.

1) By extending thread class:

•The class should extend Java Thread class.

•The class should override the run() method.

•The functionality that is expected by the Thread to be executed is written in the run() method.

void start(): Creates a new thread and makes it runnable.

void run(): The new thread begins its life inside this method.

Example:

```
public class MyThread extends Thread
{
        public void run()
        {
                System.out.println("thread is running...");
        }
        public static void main(String[] args)
        {
                MyThread obj = new MyThread();
                obj.start();
        }
}
```

Output: thread is running...

```
class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 5; i++) {
            System.out.println( i + " " + getName());
            try {
                sleep(5000);
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
    public static void main(String args[])
    {   SimpleThread st=new SimpleThread("My first Thread");
        st.start();
    }
}
```

Output:

0 My first Thread
1 My first Thread
2 My first Thread
3 My first Thread
4 My first Thread
DONE! My first Thread

•***Start():*** Creation of thread object never starts execution, we need to call 'start()' method to run a thread.

•**Sleep():** It makes current executing thread to sleep for a specified interval of time. Time is in milliseconds.

•**Yield():** It makes current executing thread object to pause temporarily and gives control to other thread to execute.

notify(): This wakes up threads that called wait() on the same object and moves the thread to ready state.

•**notifyAll():** This method is inherited from Object class. This method wakes up all threads that are waiting on this object's monitor to acquire lock.

wait(): when wait() method is invoked on an object, the thread executing that code gives up its lock on the object immediately and moves the thread to the wait state.

2) By Implementing Runnable interface:

- The class should implement the Runnable interface.
- The class should implement the run() method in the Runnable interface.
- Create an instance of the Thread class and pass your Runnable Object to its constructor as a parameter. A Thread object is created that can run your Runnable class.
- The functionality that is expected by the Thread to be executed is put in the run() method

```
class Multi3 implements Runnable
{  public void run()
   {    System.out.println("thread is running...");
    }
   public static void main(String args[])
   {
       Multi3 m1=new Multi3();
       Thread t1 =new Thread(m1);
       t1.start();
   }
}
```

Priority of a Thread (Thread Priority):

There are 3 constants defined in Thread class:

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling

1.public static int MIN_PRIORITY

2.public static int NORM_PRIORITY

3.public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
class TestMultiPriority1 extends Thread
{  public void run()
   {
     System.out.println("running thread name is:"+Thread.currentThread().getName());
     System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
   }
 public static void main(String args[])
 {       TestMultiPriority1 m1=new TestMultiPriority1();
         TestMultiPriority1 m2=new TestMultiPriority1();
         TestMultiPriority1 m3=new TestMultiPriority1();
         m1.setPriority(Thread.MIN_PRIORITY);
         m2.setPriority(Thread.NORM_PRIORITY);
         m3.setPriority(Thread.MAX_PRIORITY);
         m1.start();
         m2.start();
         m3.start();
  }
}
```

- **Thread Synchronization:**

  - When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.

  - The process by which this is achieved is called synchronization. Synchronization is the concept of the monitor(also called a semaphore).

  - For example, if multiple threads try to write within a same file then they may corrupt the data because one of the threads can override data or while one thread is opening the same file at the same time another thread might be closing the same file.

  - So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors.**

A monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time. When a thread acquires a lock, it is said to have entered the monitor.

All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.

Types of Synchronization

There are two types of synchronization

1. Process Synchronization
2. Thread Synchronization

Thread Synchronization:

- There are two types of thread synchronization

1. Mutual Exclusive
2. Inter-thread communication.

- Mutual Exclusive
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
- Cooperation (Inter-thread communication in java)

```
class Table{
void printTable(int n){//method not synchronized
    for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   }    }  }
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);  }
}
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;  }
public void run(){
t.printTable(100);  }
}
class TestSynchronization1{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();  }
}
```

Output:

```
5
100
10
200
15
300
20
400
25
500
```

Java synchronized method:

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

Concept of Lock:

Synchronization is built around an internal entity known as the lock or monitor. Every object has an lock associated with it. By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

```
class Table{
 synchronized void printTable(int n){
   for(int i=1;i<=5;i++){
     System.out.println(n*i);
     try{
      Thread.sleep(400);
     }catch(Exception e){System.out.println(e);}
   } } }
class MyThread1 extends Thread{
Table t;
MyThread1(Table t){
this.t=t;
}
public void run(){
t.printTable(5);
} }
```

```
class MyThread2 extends Thread{
Table t;
MyThread2(Table t){
this.t=t;  }
public void run(){
t.printTable(100);
} }
public class TestSynchronization2{
public static void main(String args[]){
Table obj = new Table();//only one object
MyThread1 t1=new MyThread1(obj);
MyThread2 t2=new MyThread2(obj);
t1.start();
t2.start();
} }
```

Output:

5
10
15
20
25
100
200
300
400
500

Inter-thread communication:

- **Inter-thread communication** or **Co-operation** is all about allowing synchronized threads to communicate with each other.
- Polling is usually implemented with the help of loops to check whether a particular condition is true or not. If it is true, certain action is taken. This wastes CPU times and makes the implementation inefficient.
- Cooperation is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
  - It is implemented by following methods of **Object class**:
    - final void wait() throws InterruptedException
    - final void notify()
    - final void notifyAll()

1) wait() method:

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only, otherwise it will throw exception.

2) notify() method:

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.

Syntax:

```
public final void notify()
```

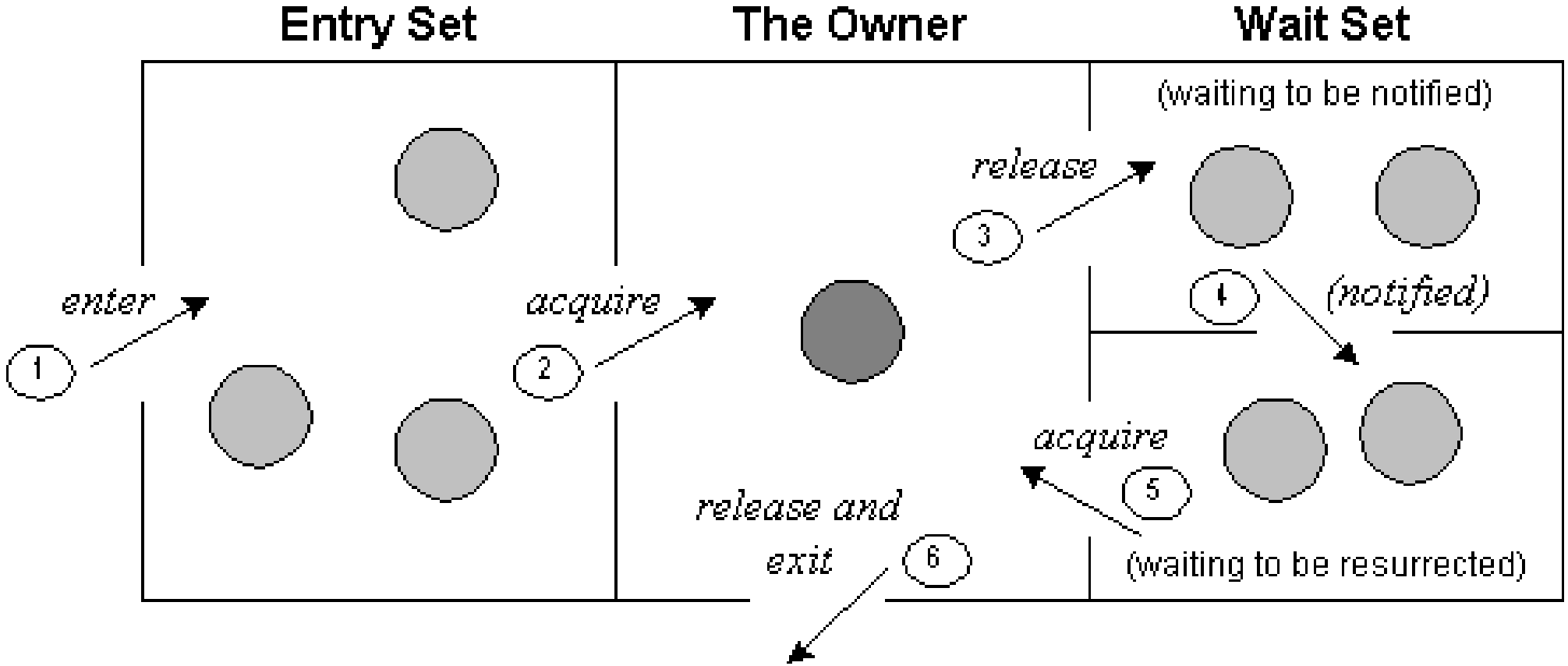
3) notifyAll() method:

Wakes up all threads that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

The following Diagram shows the process of inter-thread communication



1.Threads enter to acquire lock.
2.Lock is acquired by on thread.
3.Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.

4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5.Now thread is available to acquire lock.
6.After completion of the task, thread releases the lock and exits the monitor state of the object.

```
class Customer
{
   int amount=10000;
   synchronized void withdraw(int amount)
   {
      System.out.println("going to withdraw...");
 if(this.amount<amount)
{
System.out.println("Less balance; waiting for deposit");
    try
    {   wait();    }
     catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}
synchronized void deposit(int amount)
 {   System.out.println("going to deposit...");
     this.amount+=amount;
     System.out.println("deposit completed... ");
     notify();
  }
}
```

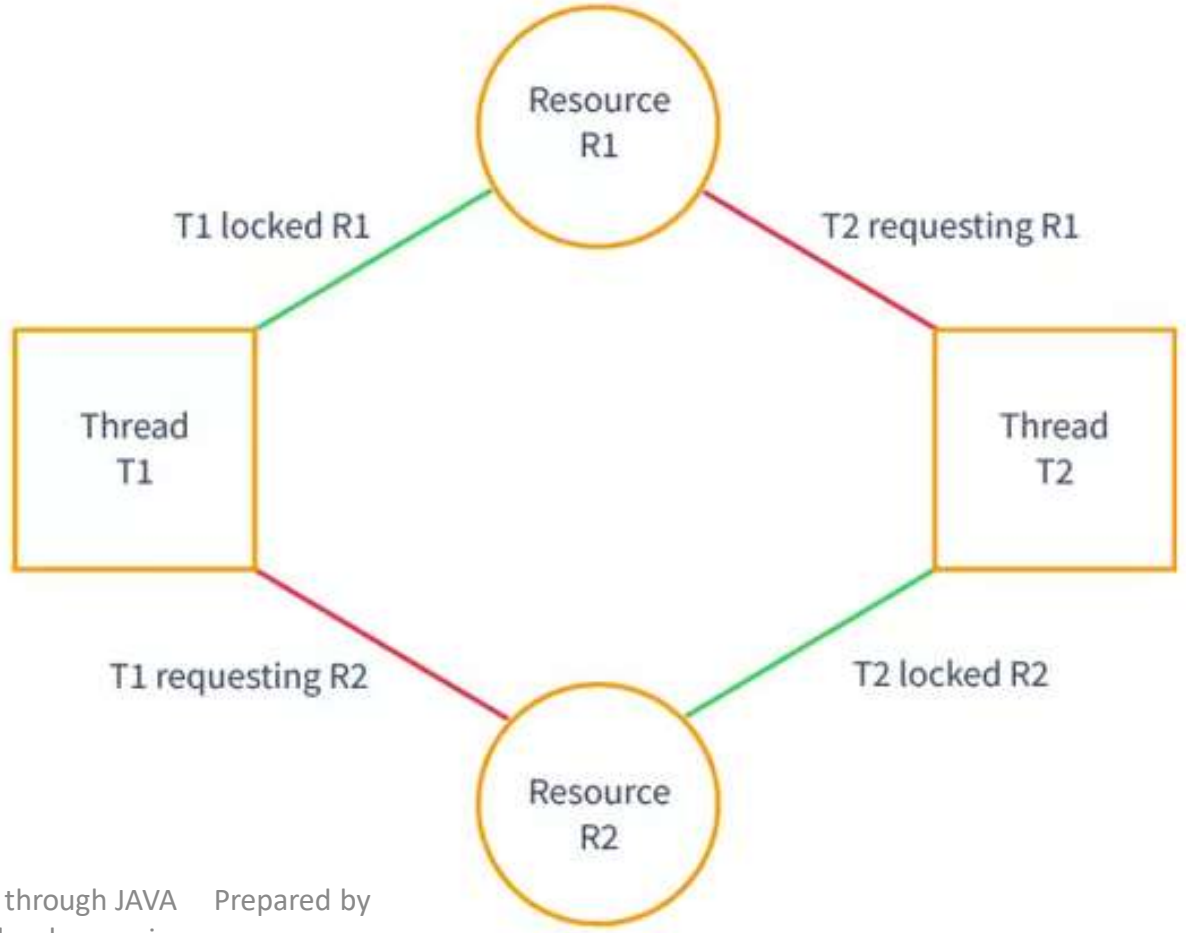
```
class Test
{
public static void main(String args[])
{
   final Customer c=new Customer();
   new Thread()
   {
      public void run()
          {
                c.withdraw(15000);
          }
    }.start();
    new Thread()
    {
                public void run()
                {

          c.deposit(10000);
                }
          }.start();
  }
}
```

Output: going to withdraw... Less balance; waiting for deposit...
going to deposit... deposit completed... withdraw completed

DEADLOCK IN JAVA

- Deadlock in Java is a situation where two or more processes wait indefinitely for one another's action. Deadlock situation takes place mainly in **Multithreaded** programming.
- **For example**, a **deadlock** can occur when the **first thread is waiting** to acquire an **object's lock** which is acquired already by the **second thread**, and the **second thread** is waiting to acquire another **object's lock** which is already acquired by the **first thread**. Hence, **both threads** are **waiting** for each other to **release the lock**, creating a **deadlock** situation.
- The following program shows how deadlock occurred in multithreading.



```
public class TestDeadlockExample1 {
 public static void main(String[] args) {
  final String resource1 = “file1";
  final String resource2 = “printer";
  // t1 tries to lock resource1 then resource2
  Thread t1 = new Thread() {
   public void run() {
     synchronized (resource1) {
      System.out.println("Thread 1: locked resource 1");
       try { Thread.sleep(100);}
       catch (Exception e) {}
      synchronized (resource2) {
      System.out.println("Thread 1: locked resource 2");
      }
    }
   }
  };
  // t2 tries to lock resource2 then resource1
   Thread t2 = new Thread() {
    public void run() {
     synchronized (resource2) {
      System.out.println("Thread 2: locked resource 2");
      try { Thread.sleep(100);}
      catch (Exception e) {}
      synchronized (resource1) {
       System.out.println("Thread 2: locked resource 1");
      }
     }
    }
   };
   t1.start();
   t2.start();
 } }
```

Output:

Thread 1: locked resource 1

Thread 2: locked resource 2

JAVA DATABASE CONNECTIVITY:

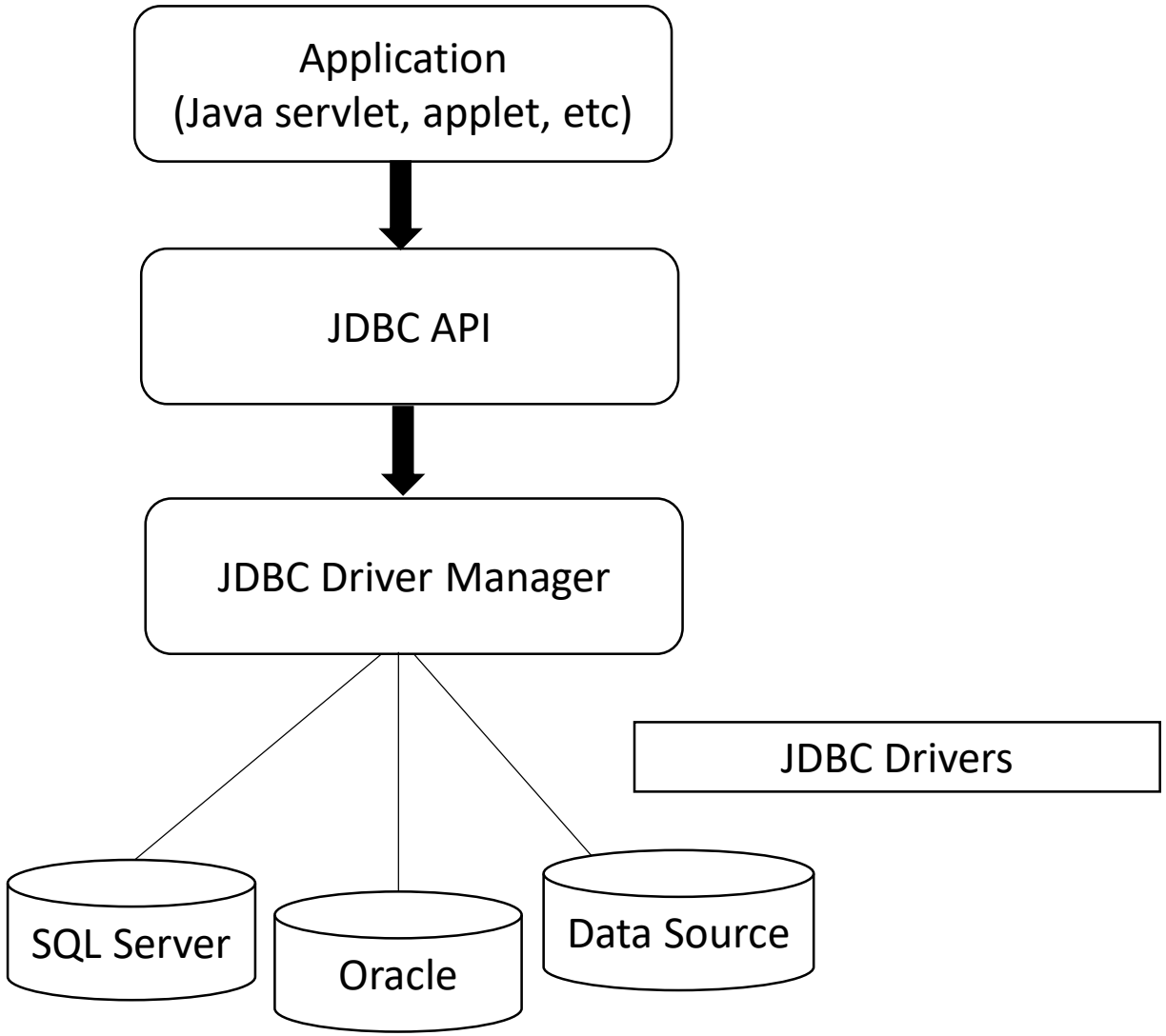
- JDBC (Java Database Database Connectivity) is a Sun Microsystems specification. The Java API is responsible for connecting to a database, issuing queries and commands, and processing database result sets.
- JDBC and database drivers operate together to access spreadsheets and databases. The design of JDBC defines the components that are utilized to connect to the database.
- The JDBC API classes and interfaces enable an application to send a request to a specific database.

Applications of JDBC:

JDBC enables you to create Java applications that handle the following three programming tasks:

- Make a connection to a data source, such as a database.
- Send database queries and update statements.
- Retrieve and process the returned database results in response to your query.

ARCHITECTURE OF JDBC



1. **Application:** It is a java **applet** or a **servlet** that communicates with a data source.

2. **The JDBC API:** The JDBC API allows Java programs to execute **SQL statements** and retrieve results. Some of the important interfaces defined in JDBC API are as follows: **Driver interface, ResultSet Interface, RowSet Interface, PreparedStatement interface, Connection interface, and Classes** defined in JDBC API are as follows: **DriverManager class, Types class, Blob class, clob class.**

3. **DriverManager:** It plays an important role in the **JDBC architecture**. It uses some database-specific drivers to connect enterprise applications to databases effectively.

4. **JDBC drivers:** To communicate with a data source through JDBC, you need a JDBC driver that intelligently communicates with the respective data source.

Types of JDBC Architecture(2-tier and 3-tier)

The JDBC architecture consists of two-tier and three-tier processing models to access a database. They are as described below:

1.Two-tier model: A Java application communicates directly to the data source. The JDBC driver enables the communication between the **application and the data source**. When a user sends a **query** to the data source, the answers for those **queries** are sent back to the user in the form of **results**. The data source can be located on a different machine on a network to which a user is connected. This is known as a **client/server configuration**, where the **user's** machine acts as a **client**, and the **machine has the data source** running acts as the **server**.

2.Three-tier model: The user's **queries** are sent to **middle-tier services**, from which the commands are again sent to the **data source**. The **results** are sent back to the **middle tier** and then to the **user**. This type of model is found very useful by management information system directors.

INSTALLING MYSQL:

- Download the **MySQL** software from the following link.

https://dev.mysql.com/downloads/installer/

After downloading, unzip it, and double-click the **MSI installer .exe** file.
Then follow the steps below:

1. **"Choosing a Setup Type" screen:** Choose "Full" setup type. This installs all MySQL products and features. Then click the "Next" button to continue.
2. **"Check Requirements" screen:** The installer checks if your **PC** has the requirements needed. If there are some failing requirements, click on each item to try to resolve them by clicking on the Execute button which will install all requirements automatically. Click "Next".
3. **"Installation" screen:** See what products that will be installed. Click "Execute" to download and install the Products. After finishing the installation, click "Next".
4. **"Product Configuration" screen:** See what products that will be configured. Click the "MySQL Server 8.0.23" option to configure the MySQL Server. Click the "Next" button. Choose the "Standalone MySQL Server/Classic MySQL Replication" option and click on the "Next" button. On page "Type and Networking" set Config Type to "Development Computer" and "Connectivity" to "TCP/IP" and "Port" to "3006". Then, click the "Next" button.
5. **"Authentication Method" screen:** Choose "Use Strong **Password Encryption** for Authentication". Click "Next".

6. **"Accounts and Roles" screen:** Set a password for the root account. Click "Next".
7. **"Windows Service" screen:** Here, you configure the Windows Service to start the server. Keep the default setup, then click "Next".
8. **"Apply Configuration" screen:** Click the "Execute" button to apply the Server configuration. After finishing, click the "Finish" button.
9. **"Product Configuration" screen:** See that the Product Configuration is completed. Keep the default setting and click on the "Next" and "Finish" buttons to complete the MySQL package installation.
10. In the next screen, you can choose to configure the Router. Click on "Next", and "Finish" and then click the "Next" button.
11. **"Connect To Server" screen:** Type in the root password (from step 6). Click the "Check" button to check if the connection is successful or not. Click on the "Next" button.
12. **"Apply Configuration" screen:** Select the options and click the "Execute" button. After finishing, click the "Finish" button.
13. **"Installation Complete" screen:** The installation is complete. Click the "Finish" button.

INSTALLING MYSQL CONNECTOR/J:

Note: You must use MySQL Connector/J version 5.1 or later.

Steps:

1.Download the MySQL Connector/J drivers at dev.mysql.com.

2.Install the .jar file and note its location for future reference.

For example, install the .jar file at C:\Program Files\MySQL\MySQL Connector J\mysql-connector-java-5.1.32-bin.jar.

JDBC - ENVIRONMENT SETUP (Windows platform):

- To start developing with JDBC, you should set up your JDBC environment by following the steps shown below.
- Install the latest version of Java on your machine. Once you have installed Java on your machine, you must set environment variables to point to the correct installation directories.
- Right-click on 'My Computer' and select 'Properties'.
- Click on the 'Environment variables' button under the 'Advanced' tab.
- Now, edit the 'Path' variable and add the path to the Java executable directory at the end of it.
- For example, if the path is currently set to C:\Windows\System32, then edit it in the following way,

  C:\Windows\System32;c:\Program Files\java\jdk\bin

Note: Download and install the same version of the JDK software jar file.

JDBC CONNECTIVITY

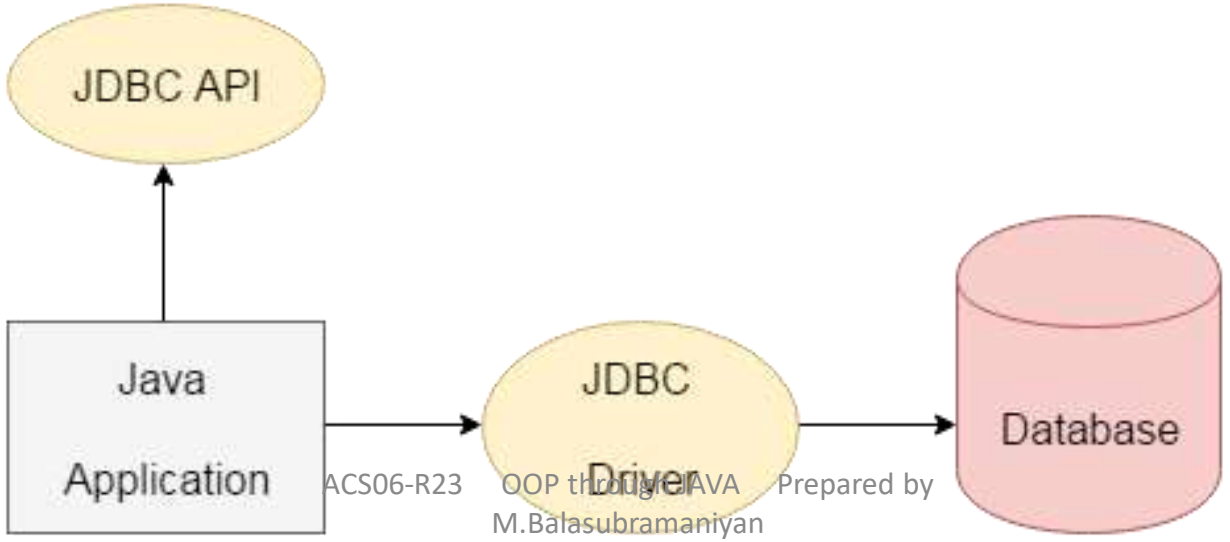
JDBC:

- JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.

There are four types of JDBC drivers:

- JDBC-ODBC Bridge Driver,
- Native Driver,
- Network Protocol Driver, and
- Thin Driver

- We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the database. It is like Open Database Connectivity (ODBC) provided by Microsoft.



Purpose of JDBC:

- Before JDBC, ODBC API was the database API to connect and execute the query with the database. But, ODBC API uses ODBC driver which is written in C language (i.e. platform dependent and unsecured). That is why Java has defined its own API (JDBC API) that uses JDBC drivers (written in Java language).

- We can use JDBC API to handle database using Java program and can perform the following activities:

1. Connect to the database
2. Execute queries and update statements to the database
3. Retrieve the result received from the database.

API Definition:

- API (Application programming interface) is a document that contains a description of all the features of a product or software. It represents classes and interfaces that software programs can follow to communicate with each other. An API can be created for applications, libraries, operating systems, etc.

WORKING OF JDBC:

Java application that needs to communicate with the database has to be programmed using JDBC API. JDBC Driver supporting data sources such as Oracle and SQL server has to be added in java application for JDBC support which can be done dynamically at run time. This JDBC driver intelligently communicates the respective data source.

Creating a simple JDBC application:

```
import java.sql.*;
class MysqlCon {
public static void main(String args[]) {
   try{    Class.forName("com.mysql.jdbc.Driver");
           Connection con=DriverManager.getConnection(
           "jdbc:mysql://localhost:3306/Employee","root",“tiger");
           Statement stmt=con.createStatement();
           ResultSet rs=stmt.executeQuery("select * from emp");
           while(rs.next())
                System.out.println(rs.getInt(1)+"  "+rs.getString(2)+"  "+rs.getString(3));
           con.close();
       }catch(Exception e){ System.out.println(e); }
} }
```

JAVA code to insert records into MySQL database.

```
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import com.mysql.jdbc.Connection;
import com.mysql.jdbc.PreparedStatement;
import com.mysql.jdbc.Statement;
public class JavaInsertDemo {
   public static void main(String[] args) {
      Connection conn = null;
      Statement stmt = null;
      try {
         try {
            Class.forName("com.mysql.jdbc.Driver");
         }
       catch (Exception e) {
            System.out.println(e);
      }
```

```
conn = (Connection) DriverManager.getConnection("jdbc:mysql://localhost/business", "Manish", "123456");
    System.out.println("Connection is created successfully:");
    stmt = (Statement) conn.createStatement();
    String query1 = "INSERT INTO InsertDemo " + "VALUES (1, 'John', 34)";
    stmt.executeUpdate(query1);
    query1 = "INSERT INTO InsertDemo " + "VALUES (2, 'Carol', 42)";
    stmt.executeUpdate(query1);
    System.out.println("Record is inserted in the table successfully.....................");
    }
    catch (SQLException excep) {
      excep.printStackTrace();
    } catch (Exception excep) {
      excep.printStackTrace();
    }
```

```
finally {
        try {
          if (stmt != null)
             conn.close();
        }
        catch (SQLException se) {}
        try {
          if (conn != null)
             conn.close();
        }
         catch (SQLException se) {
          se.printStackTrace();
        }
     }
     System.out.println("Please check it in the MySQL Table..... ……..");
   }
}
```

RESULTSET INTERFACE:

- The object of ResultSet maintains a cursor pointing to a row of a table. Initially, cursor points to before the first row.
- By default, ResultSet object can be moved forward only and it is not updatable.
- But we can make this object to move forward and backward direction by passing either TYPE_SCROLL_INSENSITIVE or TYPE_SCROLL_SENSITIVE in createStatement(int,int) method as well as we can make this object as updatable by:

1. Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
2. ResultSet.CONCUR_UPDATABLE);

Commonly used methods of ResultSet interface:

1) public boolean next():	is used to move the cursor to the one row next from the current position.
2) public boolean previous():	is used to move the cursor to the one row previous from the current position.
3) public boolean first():	is used to move the cursor to the first row in result set object.
4) public boolean last():	is used to move the cursor to the last row in result set object.
5) public boolean absolute(int row):	is used to move the cursor to the specified row number in the ResultSet object.

Simple example of ResultSet interface to retrieve the data of 3rd row.

```
import java.sql.*;
class FetchRecord
{
 public static void main(String args[])throws Exception
 {
   Class.forName("oracle.jdbc.driver.OracleDriver");
   Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:1521:xe","system","oracle");
  Statement stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,ResultSet.CONCUR_UPDATABLE);
   ResultSet rs=stmt.executeQuery("select * from emp765");
   //getting the record of 3rd row
   rs.absolute(3);
   System.out.println(rs.getString(1)+" "+rs.getString(2)+" "+rs.getString(3));
   con.close();
} }
```

JAVAFX:

- JavaFX is a Java library used to develop **Desktop applications** and **Rich Internet Applications** (RIA). The applications built in JavaFX can run on multiple platforms including **Web, Mobile, and Desktops**. JavaFX is a comprehensive set of **graphics and media packages** bundled with the Java SE Development Kit (JDK).

- A fundamental feature of JavaFX is its declarative language, **FXML**. It allows developers to define their applications' **user interface** (UI) using an **XML-based syntax**. This separation of UI from application logic simplifies the design and maintenance of complex interfaces, enhancing development efficiency.

- JavaFX applications are structured around stages and scenes. A stage represents the application's main window, while a scene defines the content within that window. Various nodes, such as **buttons, labels, text fields, etc.,** can be added to the scene to create a visually appealing and interactive UI.

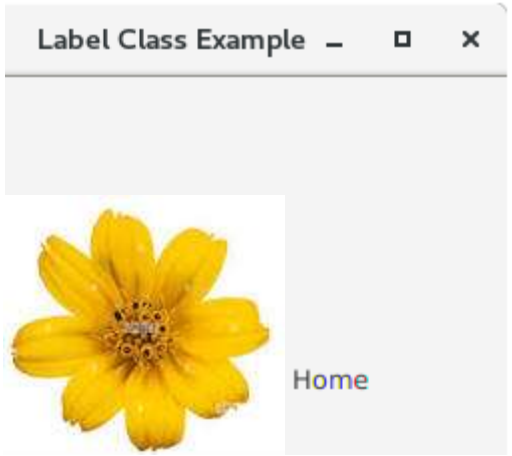
- **JavaFX** is intended to replace **Swing** in Java applications as a **GUI framework.** However, It provides more functionalities than swing. JavaFX also provides its components and doesn't depend upon the operating system. It is **lightweight and hardware-accelerated.**

FEATURES OF JAVAFX

Feature	Description
Java Library	It is a Java library which consists of many classes and interfaces that are written in Java.
FXML	FXML is the XML based Declarative mark-up language. The coding can be done in FXML to provide the more enhanced GUI to the user.
Scene Builder	Scene Builder generates FXML mark-up which can be ported to an IDE.
Web view	Web pages can be embedded with JavaFX applications. Web View uses WebKitHTML technology to embed web pages.
Built-in UI Controls	JavaFX contains built-in components that are not dependent on operating system. The UI component are just enough to develop a full featured application.
CSS like styling	JavaFX code can be embedded with the CSS to improve the style of the application. We can enhance the view of our application with the simple knowledge of CSS.
Swing interoperability	The JavaFX applications can be embedded with swing code using the Swing Node class. We can update the existing swing application with the powerful features of JavaFX.
Canvas API	Canvas API provides the methods for drawing directly in an area of a JavaFX scene.
Rich Set of APIs	JavaFX provides a rich set of API's to develop GUI applications.
Integrated Graphics Library	An integrated set of classes are provided to deal with 2D and 3D graphics.

Program to build a GUI that displays text in label and image in an ImageView

```
import java.io.FileInputStream;
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
public class LabelTest extends Application
{
    @Override
    public void start(Stage primaryStage) throws Exception {
        StackPane root = new StackPane();
        FileInputStream input= new FileInputStream("D:/Image/label.png");
        Image image = new Image(input);
        ImageView imageview=new ImageView(image);
        Label my_label=new Label("Home",imageview);
        Scene scene=new Scene(root,300,300);
        root.getChildren().add(my_label);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Label Class Example");
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
    }
}
```



JAVAFX - EVENT HANDLING

- In JavaFX, we can develop **GUI applications, web applications and graphical applications**. In such applications, whenever a user interacts with the application (nodes), an event is said to have been occurred.
- For example, **clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page** are the activities that causes an event to happen.

Types of Events:

The events can be broadly classified into the following two categories :−

- **Foreground Events** − Those events which require the **direct interaction** of a user. They are generated as consequences of a person interacting with the graphical components in a Graphical User Interface. For example, **clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page**, etc.
- **Background Events** − Those events that don't require the interaction of end-user are known as background events. The operating **system interruptions, hardware or software failure, timer expiry, operation completion** are the example of background events.

HANDLING MOUSE EVENT

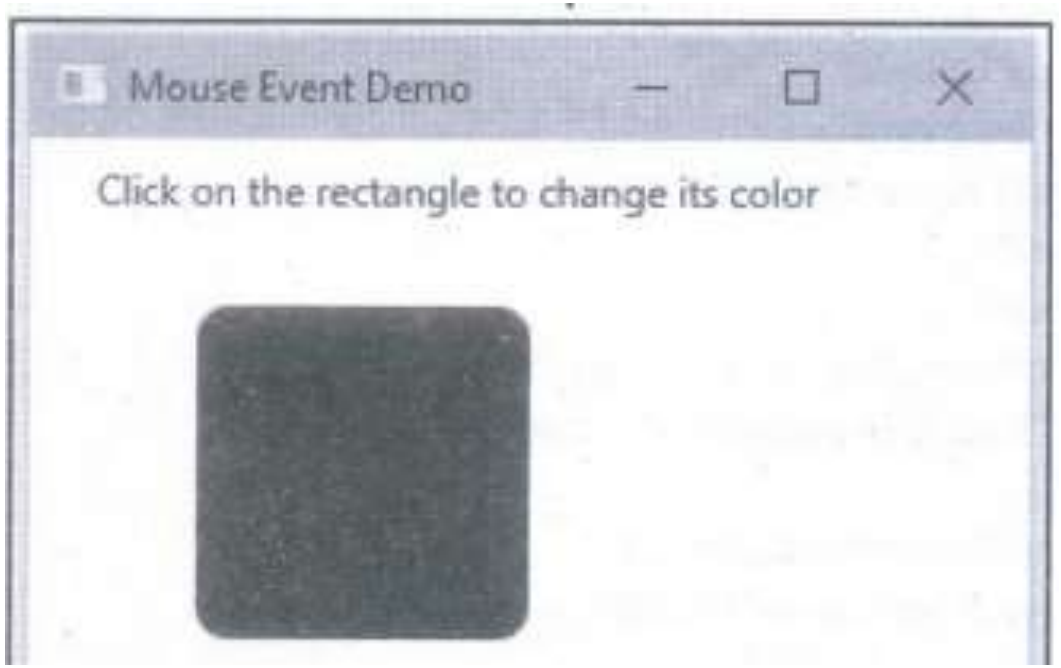
Mouse event is fired when the **mouse button is pressed, released, clicked, moved or dragged** on the node. Following table shows the user actions and associated event-handling activities -

User Action	Event Type	EventHandler Properties
Pressing, releasing, or typing a key on keyboard.	KeyEvent	onKeypressed onKeyReleased onKeyTyped
Moving, Clicking, or dragging the mouse.	MouseEvent	onMouseClicked onMouseMoved onMousePressed onMouseReleased onMouseEntered onMouseExited
Pressing, Dragging, and Releasing of the mouse button.	MouseDragEvent	onMouseDragged onMouseDragEntered onMouseDragExited onMouseDragged onMouseDragOver onMouseDragReleased

```
package myjavafxapplication;
import javafx.application. Application;
import javafx.event.EventHandler;
import javafx.scene.Group;
import javafx.scene.Scene;
import javafx.scene.input. MouseEvent;
import javafx.scene.paint.Color;
import javafx.scene.shape: Rectangle;
import javafx.scene.text.Text;
import javafx.stage.Stage;
```

```
public class MyJavaFXApplication extends Application {
@Override
public void start(Stage primaryStage) {
Rectangle rect = new Rectangle(50,50,100,100);
rect.setArcWidth(20);
rect.setArcHeight(20);
rect.setFill(Color.BLUE);
Text text = new Text (20,20,"Click on the rectangle to change its color");
//Handling the mouse event
rect.setOnMouseClicked(e->{
rect.setFill(Color.RED);
});
```

```
Group root = new Group (rect,text);
Scene scene = new Scene(root,300,200);
primaryStage.setScene(scene);
primaryStage.setTitle("Mouse Event Demo");
primaryStage.show();
}
public static void main(String[] args)
{   launch(args);
}  }
```



Program Explanation: In the above program,

(1) We have created one rectangular object and it is colored using blue color.

(2) The mouse click event is used. When the user clicks over the rectangle, the color of the rectangle gets changed to red.

(3) Mouse Event is created and handled using the following code

```
rect.setOnMouseClicked(e->{
rect.setFill(Color.RED);
});
```