Module -1

Java
Write Once, Run Anywhere

Dr Sumit Badotra

Java - General

- Java is:
  - platform independent programming language
  - similar to C++ in syntax
  - a high-level, general-purpose, object-oriented, and secure programming language developed by James Gosling at Sun Microsystems,

Java - Editions

Each edition of Java has different capabilities. There are three editions of Java:

- **Java Standard Editions (JSE):** It is used to create programs for a desktop computer.
- **Java Enterprise Edition (JEE):** It is used to create large programs that run on the server and manages heavy traffic and complex transactions.
- **Java Micro Edition (JME):** It is used to develop applications for small devices such as set-top boxes, phone, and appliances.

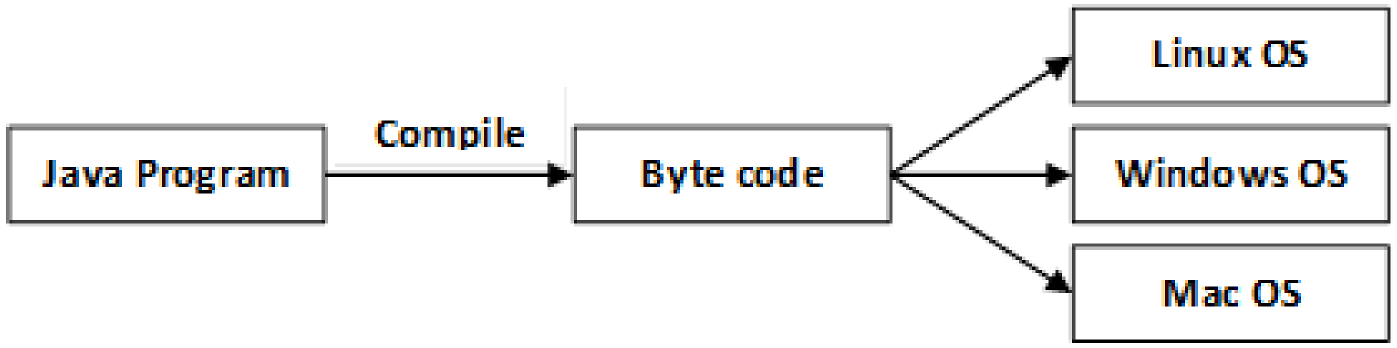
Features of Java

- Simple
- Robust (Java makes an effort to check error at run time and compile time. It uses a strong memory management system called garbage collector
- OOPs
- Multithreading
- Secure (Java contains a security manager that defines the access of Java classes)
- Platform Independent

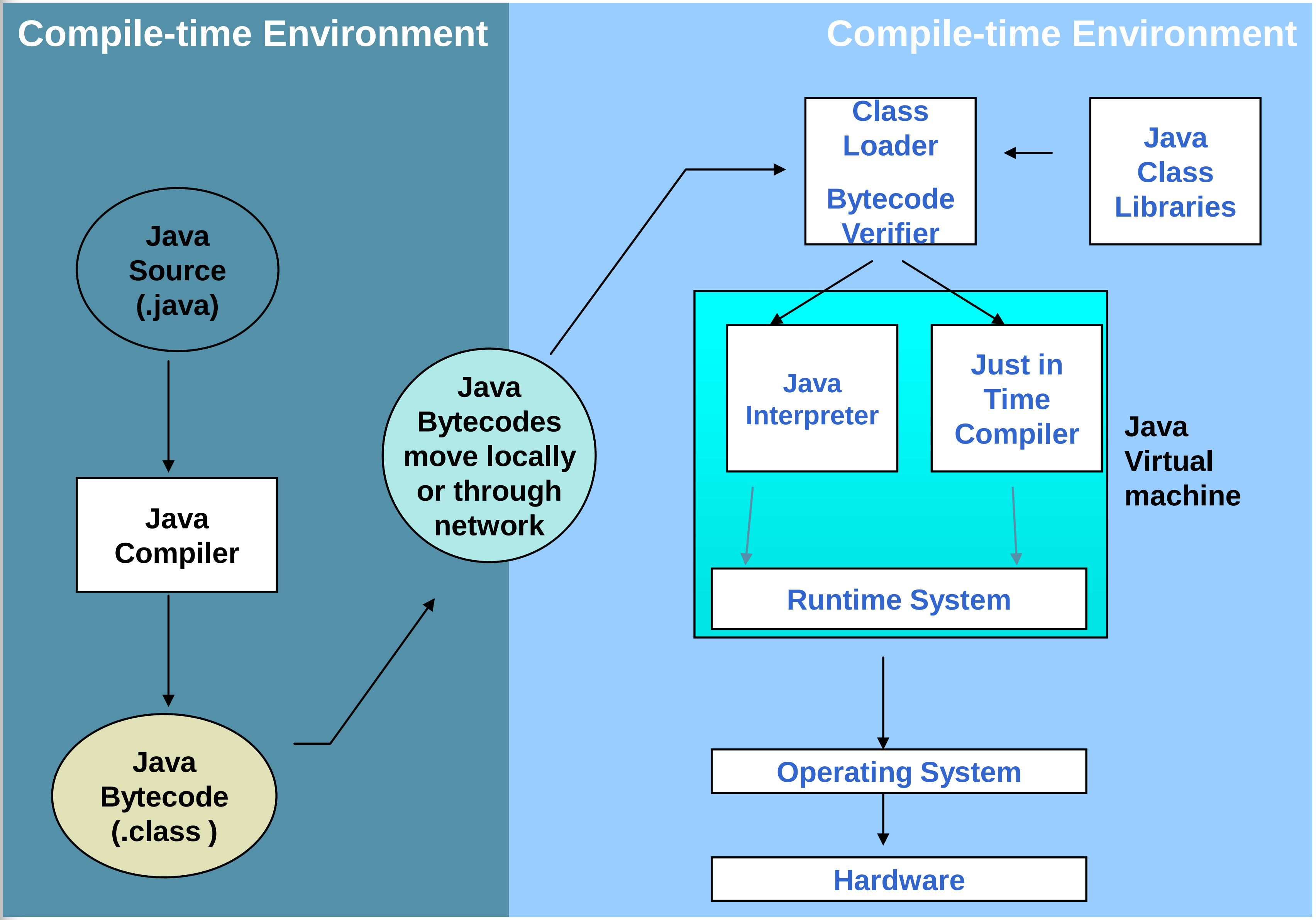
Do you Know?

Origin and Name: Java was originally developed by James Gosling and his team at Sun Microsystems in the early 1990s. It was initially named "Oak," but later the name was changed to "Java" due to trademark conflicts and the association with coffee.

Java Platform independency



How it works…!


Compile-time Environment
Compile-time Environment
Java Source (.java)
Java Compiler
Java Bytecode (.class)
Java Bytecodes move locally or through network
Class Loader
Bytecode Verifier
Java Class Libraries
Java Interpreter
Just in Time Compiler
Java Virtual machine
Runtime System
Operating System
Hardware

JVM

- The Java Virtual Machine (JVM) is an integral part of the Java Runtime Environment (JRE) and serves as the runtime engine for Java applications.
- It is a virtual machine that enables the execution of Java bytecode, which is a compiled form of Java source code.
- The primary purpose of the JVM is to provide a platform-independent and secure environment for running Java programs.

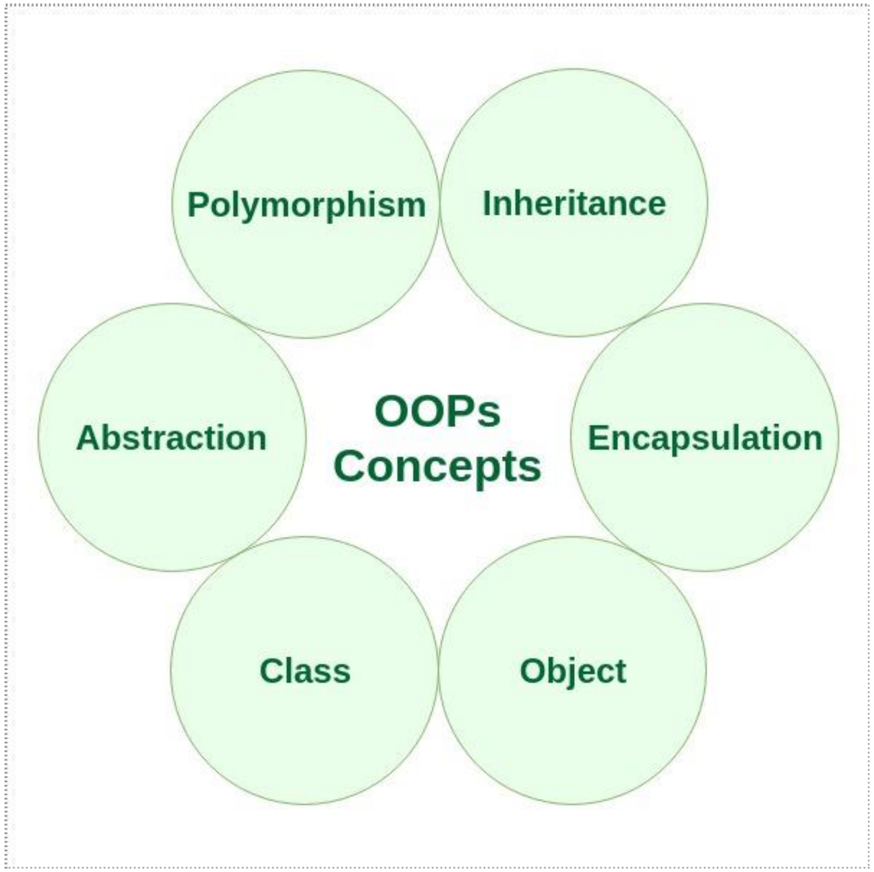
How it works…!

- Java is independent only for one reason:
  - Only depends on the Java Virtual Machine (JVM),
  - code is compiled to *bytecode*, which is interpreted by the resident JVM,
  - JIT (just in time) compilers attempt to increase speed.

Java Advantages

- Portable - Write Once, Run Anywhere
- Security has been well thought through
- Robust memory management
- Designed for network programming
- Multi-threaded (multiple simultaneous tasks)
- Dynamic & extensible (loads of libraries)
  - Classes stored in separate files
  - Loaded only when needed

OOPs in Java


Polymorphism
Inheritance
Abstraction
OOPs Concepts
Encapsulation
Class
Object

Object-Oriented

- Java supports OOD
  - Polymorphism
  - Inheritance
  - Encapsulation
- Java programs contain nothing but definitions and instantiations of classes
  - Everything is encapsulated in a class!

Basic Java Syntax

Example

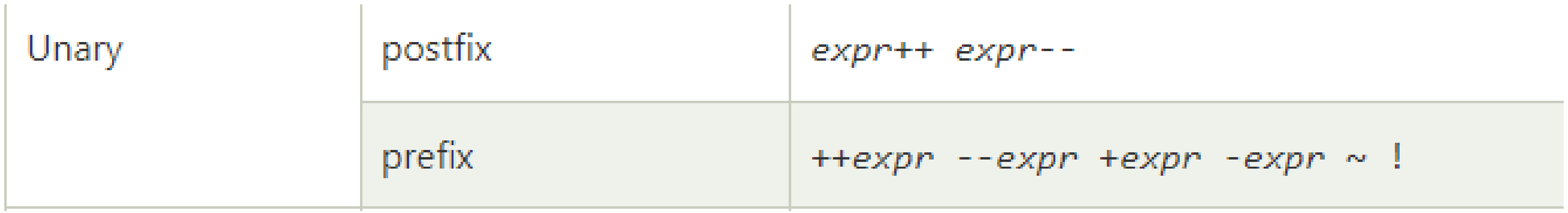
- **Simple.java**

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

Java Operators

- **Operator** in Java is a symbol that is used to perform operations. For example: +, -, *, / etc.
- There are many types of operators in Java which are given below:
- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Unary Operator



Unary	postfix	*expr++ expr--*
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

•incrementing/decrementing a value by one
•negating an expression
•inverting the value of a boolean

Example

- public class OperatorExample{
- public static void main(String args[]){
- int x=10;
- System.out.println(x++);
- System.out.println(++x);
- System.out.println(x--);
- System.out.println(--x);
- }}

Example 2

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

}}
```

Example 3

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

- boolean c=true;
- boolean d=false;

- System.out.println(!c);//false (opposite of boolean value)
- System.out.println(!d);//true
- }}

Arithmetic Operator

- Java arithmetic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Example

- `public class OperatorExample{`
- `public static void main(String args[]){`
- `int a=10;`
- `int b=5;`
- `System.out.println(a+b);//15`
- `System.out.println(a-b);//5`
- `System.out.println(a*b);//50`
- `System.out.println(a/b);//2`
- `System.out.println(a%b);//0`
- `}}`

Statements & Blocks

- public class OperatorExample{
- public static void main(String args[]){
- System.out.println(10*10/5+3-1*4/2);
- }}

What is the output???

Java Left Shift Operator

- The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Example

- public class OperatorExample{
- public static void main(String args[]){
- System.out.println(10<<2);//10*2^2=10*4=40
- System.out.println(10<<3);//10*2^3=10*8=80
- System.out.println(20<<2);//20*2^2=20*4=80
- System.out.println(15<<4);//15*2^4=15*16=240
- }}

Java Right Shift Operator

- The Java right shift operator >> is used to move the value of the left operand to right by the number of bits specified by the right operand.

- public OperatorExample{
- public static void main(String args[]){
- System.out.println(10>>2);//10/2^2=10/4=2
- System.out.println(20>>2);//20/2^2=20/4=5
- System.out.println(20>>3);//20/2^3=20/8=2
- }}

Java Assignment Operator

- Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

- public class OperatorExample{
- public static void main(String args[]){
- int a=10;
- int b=20;
- a+=4;//a=a+4 (a=10+4)
- b-=4;//b=b-4 (b=20-4)
- System.out.println(a);
- System.out.println(b);
- }}

Example

```
public class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Java Ternary Operator

- Java Ternary operator is used as one line replacement for if-then-else statement and used a lot in Java programming.

- It is the only conditional operator which takes three operands.

Examples

1. public class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}

```
1. public                  class
   OperatorExample{
2. public    static    void
   main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(m
   in);
7. }}
```

Java Bitwise Operator

- The bitwise | operator always checks both conditions whether first condition is true or false.
- The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

Example

- public class OperatorExample{
- public static void main(String args[]){
- int a=10;
- int b=5;
- int c=20;
- System.out.println(a>b||a<c);//true || true = true
- System.out.println(a>b|a<c);//true | true = true
- //|| vs |
- System.out.println(a>b||a++<c);//true || true = true
- System.out.println(a);//10 because second condition is not checked
- System.out.println(a>b|a++<c);//true | true = true
- System.out.println(a);//11 because second condition is checked
- }}

Module -1

Decision Making in Java (if, if-else, switch, break, continue, jump, while, For, do-while,)

Dr. Sumit Badotra

Java - General

- Decision Making in programming is similar to decision-making in real life.
- In programming also face some situations where we want a certain block of code to be executed when some condition is fulfilled.

Java - Selection

Java’s Selection statements:

- if
- if-else
- nested-if
- if-else-if
- switch-case
- jump – break, continue, return

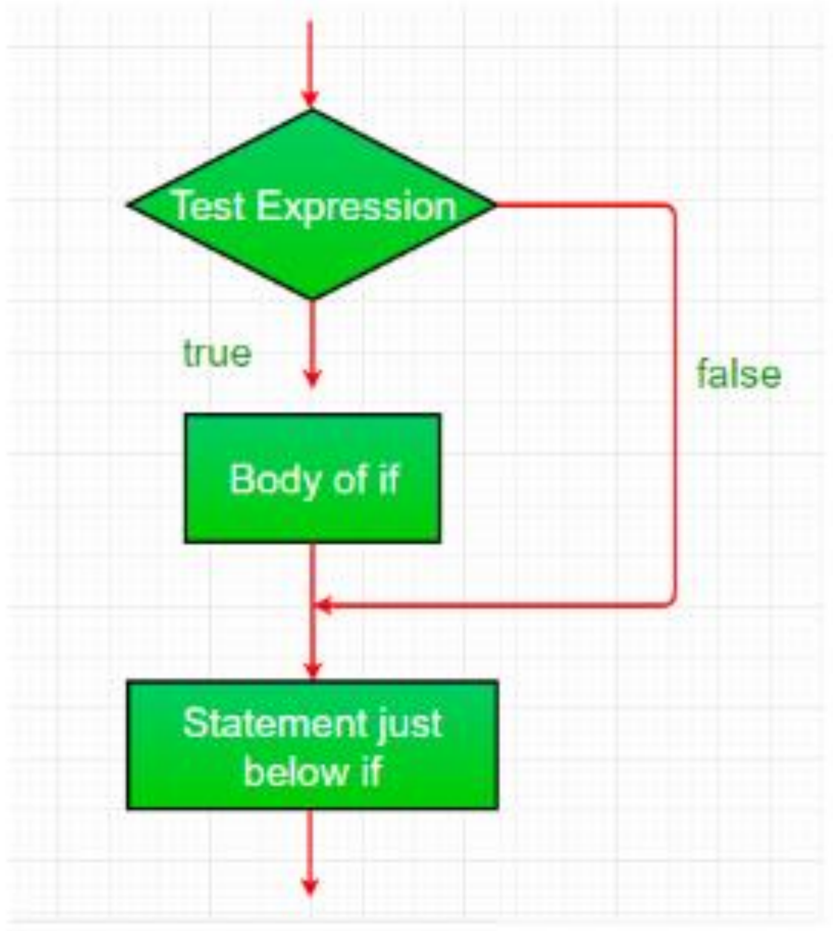
if

- if statement is the most simple decision-making statement.
- It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statements is executed otherwise not.

Syntax

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

If Condition



Util Package?

util. * is a built-in package in Java which encapsulates a similar group of classes, sub-packages and interfaces.

The * lets you import a class from existing packages and use it in the program as many times you need.

Example

```
import java.util.Scanner;  // Import the Scanner class

class Main {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

How it works…!

```
import java.util.*;

class IfDemo {
		public static void main(String args[])
		{
				int i = 10;

				if (i < 15)
						System.out.println("Inside If block"); // part of if block(immediate
one statement after if condition)
						System.out.println("10 is less than 15"); //always executes as it
is outside of if block
				// This statement will be executed
				// as if considers one statement by default again below statement is outside
of if block
				System.out.println("I am Not in if");
		}
}
```

If-else

- if-else: The if statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won’t.
- But what if we want to do something else if the condition is false? Here comes the else statement. We can use the else statement with the if statement to execute a block of code when the condition is false.

Syntax

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

Example

```
// Java program to illustrate if-else statement
import java.util.*;

class IfElseDemo {
        public static void main(String args[])
        {
                int i = 10;

                if (i < 15)
                        System.out.println("i is smaller than 15");
                else
                        System.out.println("i is greater than 15");
        }
}
```

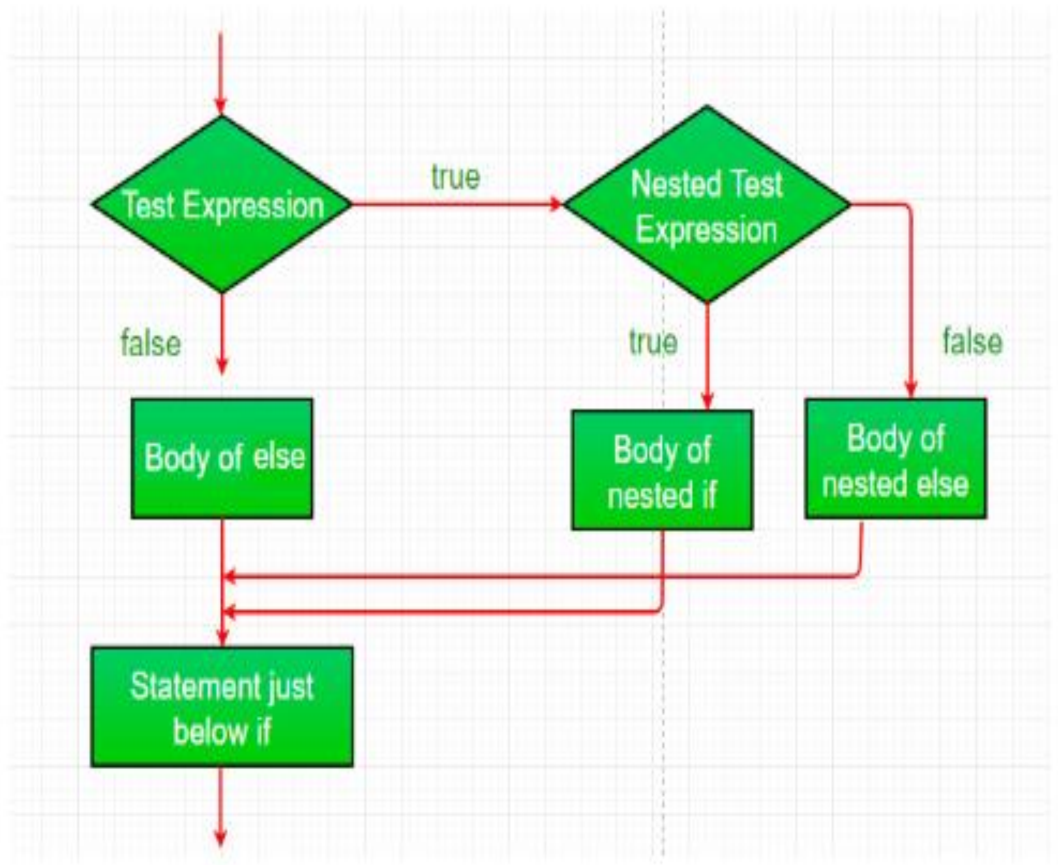
Nested if

- nested-if: A nested if is an if statement that is the target of another if or else. Nested if statements mean an if statement inside an if statement.
- Yes, java allows us to nest if statements within if statements. i.e, we can place an if statement inside another if statement.

Syntax

```
if (condition1)
{
   // Executes when condition1 is true
   if (condition2)
   {
      // Executes when condition2 is true
   }
}
```

Working


Test Expression
true
Nested Test Expression
false
true
false
Body of else
Body of nested if
Body of nested else
Statement just below if

Example

```
import java.util.*;

class NestedIfDemo {
        public static void main(String args[])
        {
                int i = 10;

                if (i == 10 || i<15) {
                        // First if statement
                        if (i < 15)
                                System.out.println("i is smaller than 15");

                        // Nested - if statement
                        // Will only be executed if statement above
                        // it is true
                        if (i < 12)
                                System.out.println(
                                        "i is smaller than 12 too");
                } else{
                                System.out.println("i is greater than 15");
                }
        }
}
```

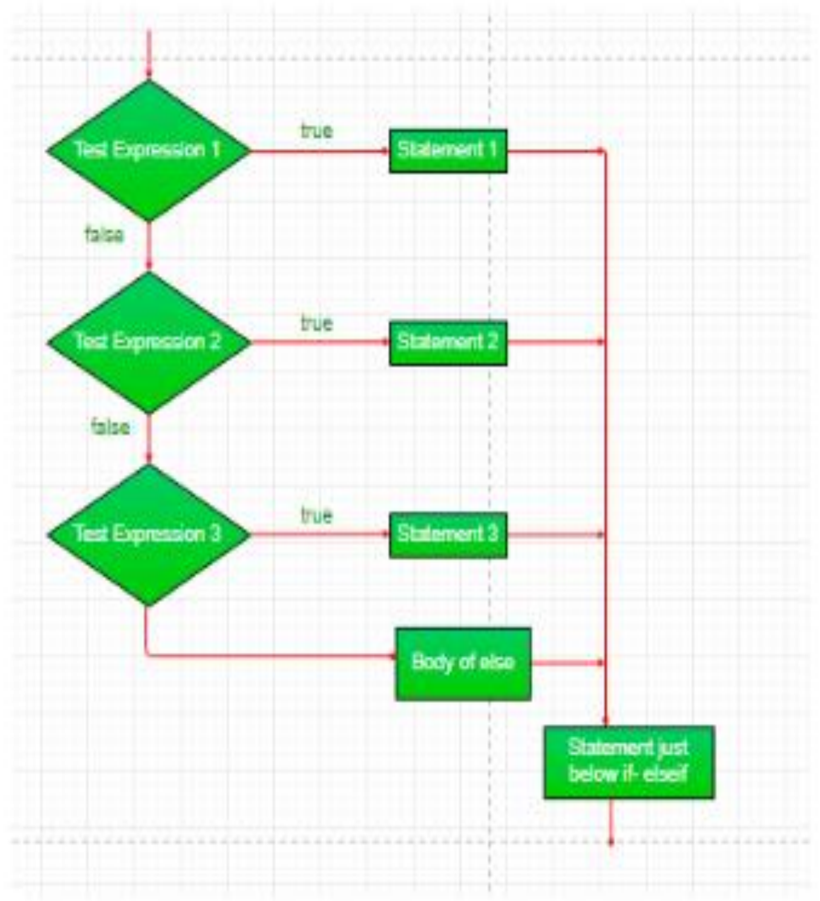
If-else-if

- **if-else-if ladder:** Here, a user can decide among multiple options.The if statements are executed from the top down.
- As soon as one of the conditions controlling the if is true, the statement associated with that 'if' is executed, and the rest of the ladder is bypassed.
- If none of the conditions is true, then the final else statement will be executed.
- There can be as many as 'else if' blocks associated with one 'if' block but only one 'else' block is allowed with one 'if' block.

Syntax

- if (condition)
-     statement;
- else if (condition)
-     statement;
- else
-     statement;

Working


Test Expression 1
true
Statement 1
false
Test Expression 2
true
Statement 2
false
Test Expression 3
true
Statement 3
Body of else
Statement just below if- elseif

Example

```
import java.util.*;

class ifelseifDemo {
        public static void main(String args[])
        {
                int i = 20;

                if (i == 10)
                        System.out.println("i is 10");
                else if (i == 15)
                        System.out.println("i is 15");
                else if (i == 20)
                        System.out.println("i is 20");
                else
                        System.out.println("i is not present");
        }
}
```

Switch-case

- **switch-case:** The switch statement is a multiway branch statement.
- It provides an easy way to dispatch execution to different parts of code based on the value of the expression.

Syntax

```
switch (expression)
{
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    break;

  case valueN:
    statementN;
    break;
  default:
    statementDefault;
}
```

Example

```
import java.io.*;

class GFG {
        public static void main (String[] args) {
                int num=20;
                switch(num){
                case 5 : System.out.println("It is 5");
                                                break;
                case 10 : System.out.println("It is 10");
                                                break;
                case 15 : System.out.println("It is 15");
                                                break;
                case 20 : System.out.println("It is 20");
                                                break;
                default: System.out.println("Not present");

                }
        }
}
```

Jump Statements

jump: Java supports three jump statements:

- break,
- continue
- and return.

- These three statements transfer control to another part of the program.

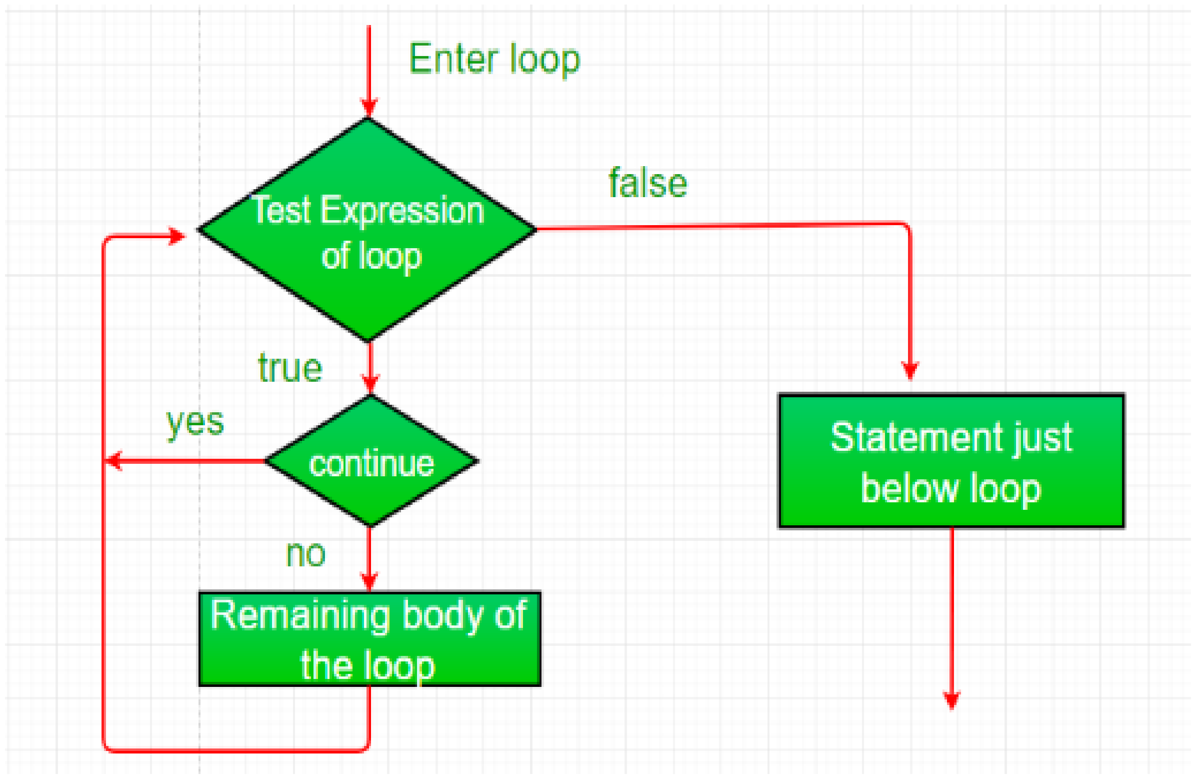
Break

- **Break:** In Java, a break is majorly used for:
  - Terminate a sequence in a switch statement (discussed earlier).
  - To exit a loop.
  - Used as a “civilized” form of goto.

Continue

- **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.

Working


Enter loop
Test Expression of loop
false
true
yes
continue
no
Remaining body of the loop
Statement just below loop

Example

```
import java.util.*;

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 + " ");
                }
        }
}
```

Return Statement

- Return: The return statement is used to explicitly return from a method.
- That is, it causes program control to transfer back to the caller of the method.

Example

```
// Java program to illustrate using return
import java.util.*;

public class Return {
    public static void main(String args[])
    {
            boolean t = true;
            System.out.println("Before the return.");

            if (t)
                    return;

            // Compiler will bypass every statement
            // after return
            System.out.println("This won't execute.");
    }
}
```

while, For, do-while,

1. The Java do-while loop is used to iterate a part of the program repeatedly, until the specified condition is true.
2. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use a do-while loop.

Syntax

1. do{
2. //code to be executed / loop body
3. //update statement
4. }while (condition);

Do-while example

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

Java Infinitive do-while Loop

- If you pass true in the do-while loop, it will be infinitive do-while loop.

Syntax

- do{
- //code to be executed
- }while(true);

Java Infinitive do-while Loop Example

```
public class DoWhileExample2 {
public static void main(String[] args) {
    do{
        System.out.println("infinitive do while loop");
    }while(true);
}
}
```

Module -1

Object Oriented concepts

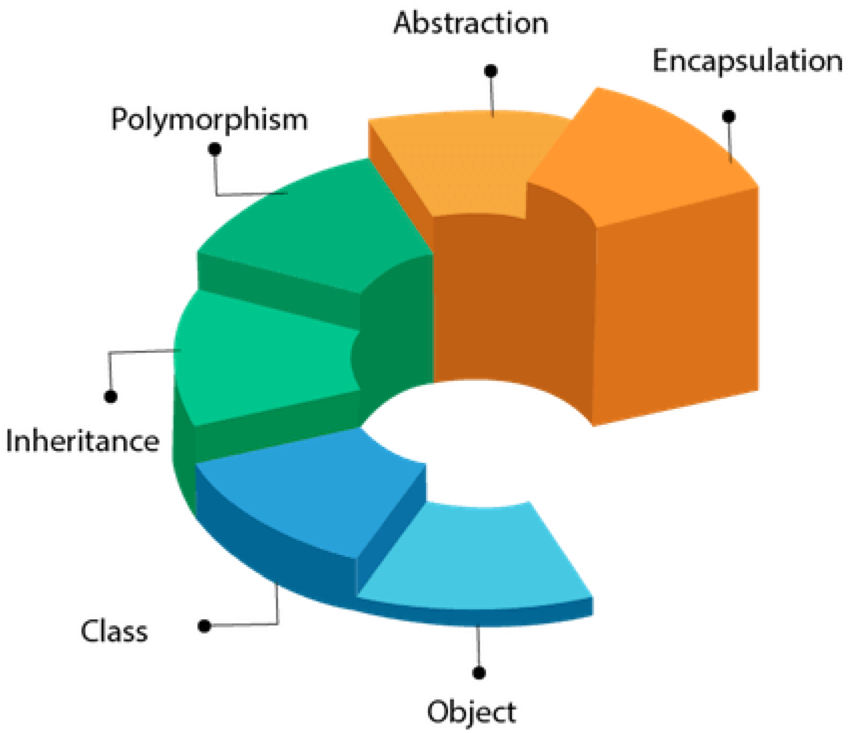
Dr. Sumit Badotra

OOPs (Object-Oriented Programming System)

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

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

Java -OOPs


Abstraction
Encapsulation
Polymorphism
Inheritance
Class
Object

What is Object?

- Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

What is Class?

- Collection of objects is called class. It is a logical entity.

- A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Example

```
// Define a simple class called "Person"
class Person {
    String name;
    int age;
    void introduce() {
        System.out.println("Hello, my name is " + name + " and I am " + age + " years old.");
    }
}
public class Main {
    public static void main(String[] args) {
        // Creating objects of the Person class
        Person person1 = new Person();
        person1.name = "Alice";
        person1.age = 30;

        Person person2 = new Person();
        person2.name = "Bob";
        person2.age = 25;

        // Calling methods on the objects
        person1.introduce(); // Output: Hello, my name is Alice and I am 30 years old.
        person2.introduce(); // Output: Hello, my name is Bob and I am 25 years old.
    }
```

Example 2

```
// Define a class called "Rectangle"
class Rectangle {
    double length;
    double width;
    double calculateArea() {
        return length * width;
    }
    double calculatePerimeter() {
        return 2 * (length + width); }}
public class Main {
    public static void main(String[] args) {
        // Creating objects of the Rectangle class
        Rectangle rectangle1 = new Rectangle();
        rectangle1.length = 5.0;
        rectangle1.width = 3.0;

        Rectangle rectangle2 = new Rectangle();
        rectangle2.length = 7.0;
        rectangle2.width = 4.0;
        // Calculating and displaying properties of the rectangles
        System.out.println("Rectangle 1 Area: " + rectangle1.calculateArea()); // Output: Rectangle 1 Area: 15.0
        System.out.println("Rectangle 1 Perimeter: " + rectangle1.calculatePerimeter()); // Output: Rectangle 1
Perimeter: 16.0

        System.out.println("Rectangle 2 Area: " + rectangle2.calculateArea()); // Output: Rectangle 2 Area: 28.0
        System.out.println("Rectangle 2 Perimeter: " + rectangle2.calculatePerimeter()); // Output: Rectangle 2
Perimeter: 22.0
    }
}
```

What is Inheritance?

- When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.
- It provides code reusability.
- It is used to achieve runtime polymorphism.

What is Polymorphism

- If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc.
- In Java, we use method overloading and method overriding to achieve polymorphism.

What is Method Overloading

```
class MathOperations {
    int add(int a, int b) {
        return a + b;
    }
    double add(double a, double b) {
        return a + b; }}
```

Method overloading refers to defining multiple methods in the same class with the same name but different parameter lists (number, type, or both).

What is Method Overriding

```
class Shape {
    void draw() {
        System.out.println("Drawing a shape");
    }
}
class Circle extends Shape {
    @Override
    void draw() {
        System.out.println("Drawing a circle"); }}
```

- *Method overriding involves creating a new implementation of an existing method in a subclass. The method in the subclass has the same name, return type, and parameters as the method in the parent class.*

What is Abstraction?

- Hiding internal details and showing functionality is known as abstraction. For example phone call, we don't know the internal processing.
- In Java, we use abstract class and interface to achieve abstraction.

What is Encapsulation?

- Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines.
- A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

What is Coupling?

- Coupling refers to the knowledge or information or dependency of another class. It arises when classes are aware of each other.
- If a class has the details information of another class, there is strong coupling. In Java, we use private, protected, and public modifiers to display the visibility level of a class, method, and field.
- You can use interfaces for the weaker coupling because there is no concrete implementation.

What is Cohesion?

- Cohesion refers to the level of a component which performs a single well-defined task. A single well-defined task is done by a highly cohesive method. The weakly cohesive method will split the task into separate parts.
- The java.io package is a highly cohesive package because it has I/O related classes and interface. However, the java.util package is a weakly cohesive package because it has unrelated classes and interfaces.

Need of Cohesion?

- Cohesion defines how the classes in Java are designed. Cohesion in Java is the principle of Object-Oriented programming.
- Cohesion is closely related to ensuring that the purpose for which a class is getting created in Java is well-focused and single.
- High cohesion ensures that each class or module has a clear and specific purpose, which leads to better organization, easier maintenance, and improved code readability.

Example of Non-Cohesive program

```
class MixedFunctionality {
    private int[] numbers;
    public MixedFunctionality(int[] numbers) {
        this.numbers = numbers;}
    // Method 1: Calculate the sum of the numbers
    public int calculateSum() {
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        return sum;}
    // Method 2: Print the numbers in reverse order
    public void printReversedNumbers() {
        for (int i = numbers.length - 1; i >= 0; i--) {
            System.out.print(numbers[i] + " ");
        }
        System.out.println();
    // Method 3: Check if the array contains a specific number
    public boolean containsNumber(int target) {
        for (int num : numbers) {
            if (num == target) {
                return true;
        return false;
public class Main {
    public static void main(String[] args) {
        int[] numArray = { 3, 7, 1, 9, 5 };
        MixedFunctionality mixedFunctionality = new MixedFunctionality(numArray);

        int sum = mixedFunctionality.calculateSum();
        System.out.println("Sum: " + sum);

        mixedFunctionality.printReversedNumbers();

 boolean containsSeven = mixedFunctionality.containsNumber(7);
        System.out.println("Contains 7: " + containsSeven);
    }
```

Example of Cohesive program

```
class NumberArray {
    private int[] numbers;

    public NumberArray(int[] numbers) {
        this.numbers = numbers;

    public int[] getNumbers() {
        return numbers;

class NumberCalculator {
    public int calculateSum(int[] numbers) {
        int sum = 0;
        for (int num : numbers) {
            sum += num;
        }
        return sum;

class NumberPrinter {
    public void printReversedNumbers(int[] numbers) {
        for (int i = numbers.length - 1; i >= 0; i--) {
            System.out.print(numbers[i] + " ");
        }
        System.out.println();


class NumberChecker {
    public boolean containsNumber(int[] numbers, int target) {
        for (int num : numbers) {
            if (num == target) {
                return true;

        return false;
public class Main {
    public static void main(String[] args) {
        int[] numArray = { 3, 7, 1, 9, 5 };

        NumberArray numberArray = new NumberArray(numArray);
        NumberCalculator calculator = new NumberCalculator();
        NumberPrinter printer = new NumberPrinter();
```

What is Association?

- Association represents the relationship between the objects. Here, one object can be associated with one object or many objects. There can be four types of association between the objects:

- One to One
- One to Many
- Many to One, and
- Many to Many

Association can be undirectional or bidirectional.

What is Aggregation?

- Aggregation is a way to achieve Association. Aggregation represents the relationship where one object contains other objects as a part of its state.
- It represents the weak relationship between objects.
- It is also termed as a has-a relationship in Java. Like, inheritance represents the is-a relationship.
- It is another way to reuse objects.

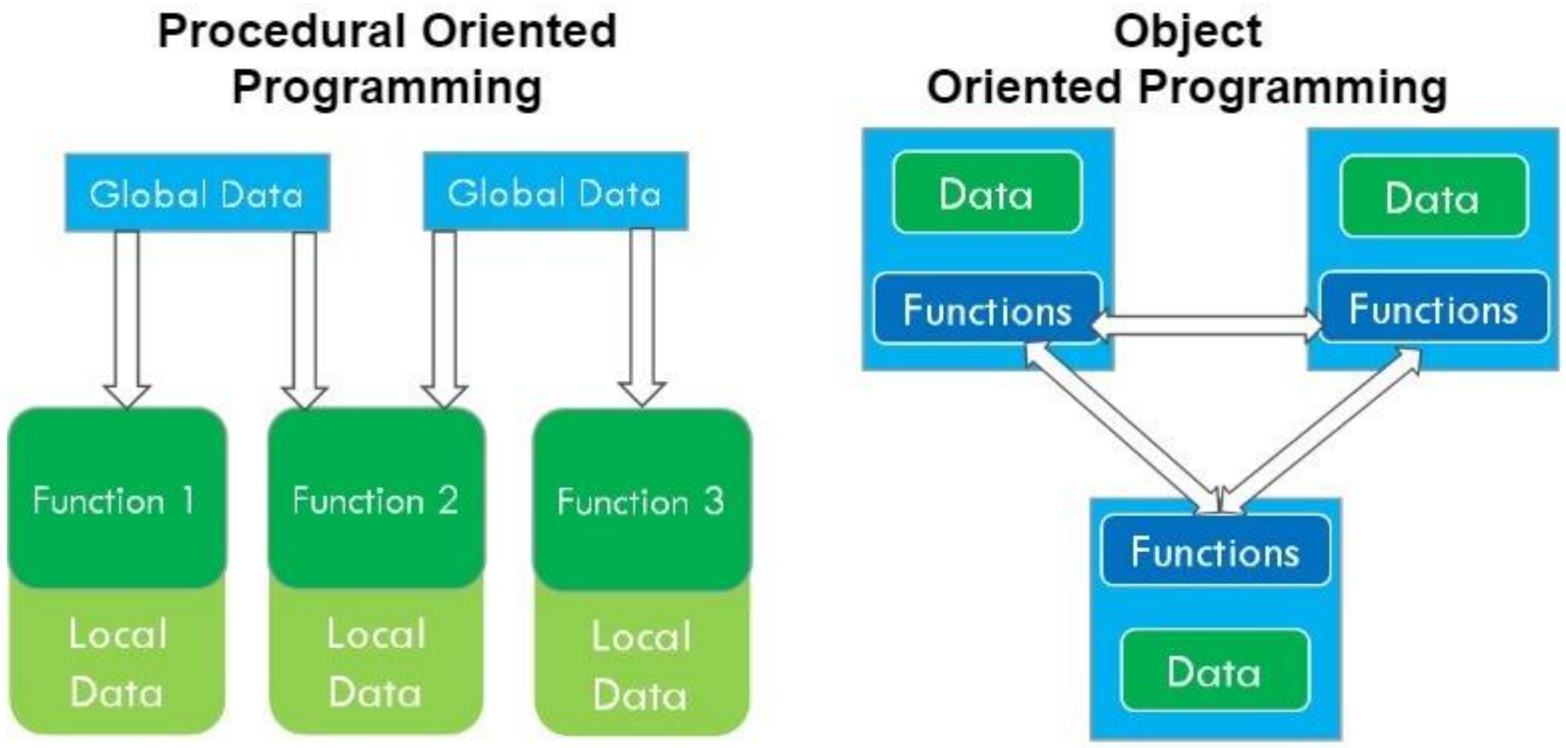
What is Composition?

- The composition is also a way to achieve Association. The composition represents the relationship where one object contains other objects as a part of its state.
- There is a strong relationship between the containing object and the dependent object. It is the state where containing objects do not have an independent existence.
- If you delete the parent object, all the child objects will be deleted automatically.

Procedural Language

- In Procedural programming, the program is divided into small functions. In Object-Oriented Programming, a program is divided into small objects.
- Some common examples of Procedural Programming are C, Fortran, VB, and Pascal. The examples of Object Oriented Programmin`g languages are Java, C++, VB.NET, Python, and C#.NET.

What is the difference?



OOPs Offer

- **Modularity:** Code is organized into objects, enhancing clarity and maintainability.
- **Reusability:** Classes and inheritance enable code reuse, reducing redundancy.
- **Abstraction:** Focuses on essential properties, making code more intuitive.
- **Polymorphism:** Objects of different classes can be treated uniformly.
- **Maintenance:** Changes are localized to objects, reducing unintended side effects.
- **Collaboration:** Facilitates team development with reduced code conflicts.
- **Real-World Mapping:** Models real-world concepts, enhancing problem domain alignment.
- **Security:** Controls data access through private and protected modifiers.
- **Simplicity:** Mirrors natural problem-solving, leading to more intuitive solutions.

Object Lifecycle

1. **Creation:** Object is born, memory allocated, attributes initialized.
2. **Usage:** Object actively used, methods and attributes utilized.
3. **References:** Object referenced by variables, stays in scope.
4. **Dereference:** Object no longer referenced, becomes eligible for cleanup.
5. **Garbage Collection:** JVM reclaims memory, removes unused objects.
6. **Finalization:** Optional cleanup before object removal (not commonly used).

Example

```
class Person {
    String name;

    Person(String name) {
        this.name = name;
        System.out.println(name + " is created.");
    }

    void introduce() {
        System.out.println("Hello, my name is " + name + ".");
    }

    protected void finalize() throws Throwable {
        System.out.println(name + " is being finalized.");
        super.finalize();}}

public class Main {
    public static void main(String[] args) {
        Person person1 = new Person("Alice"); // Creation
        person1.introduce(); // Usage

        Person person2 = new Person("Bob"); // Creation
        person2.introduce(); // Usage

        person1 = null; // Dereference
        System.gc(); // Suggesting garbage collection

        person2.introduce(); // Usage

        // At the end of the program, when it's about to exit:
        // Garbage collector may finalize person1 and person2
    }
}
```

Module -1

Wrapper classes (Boolean, Integer, Double, Character)

Dr. Sumit Badotra

Wrapper Class

- 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

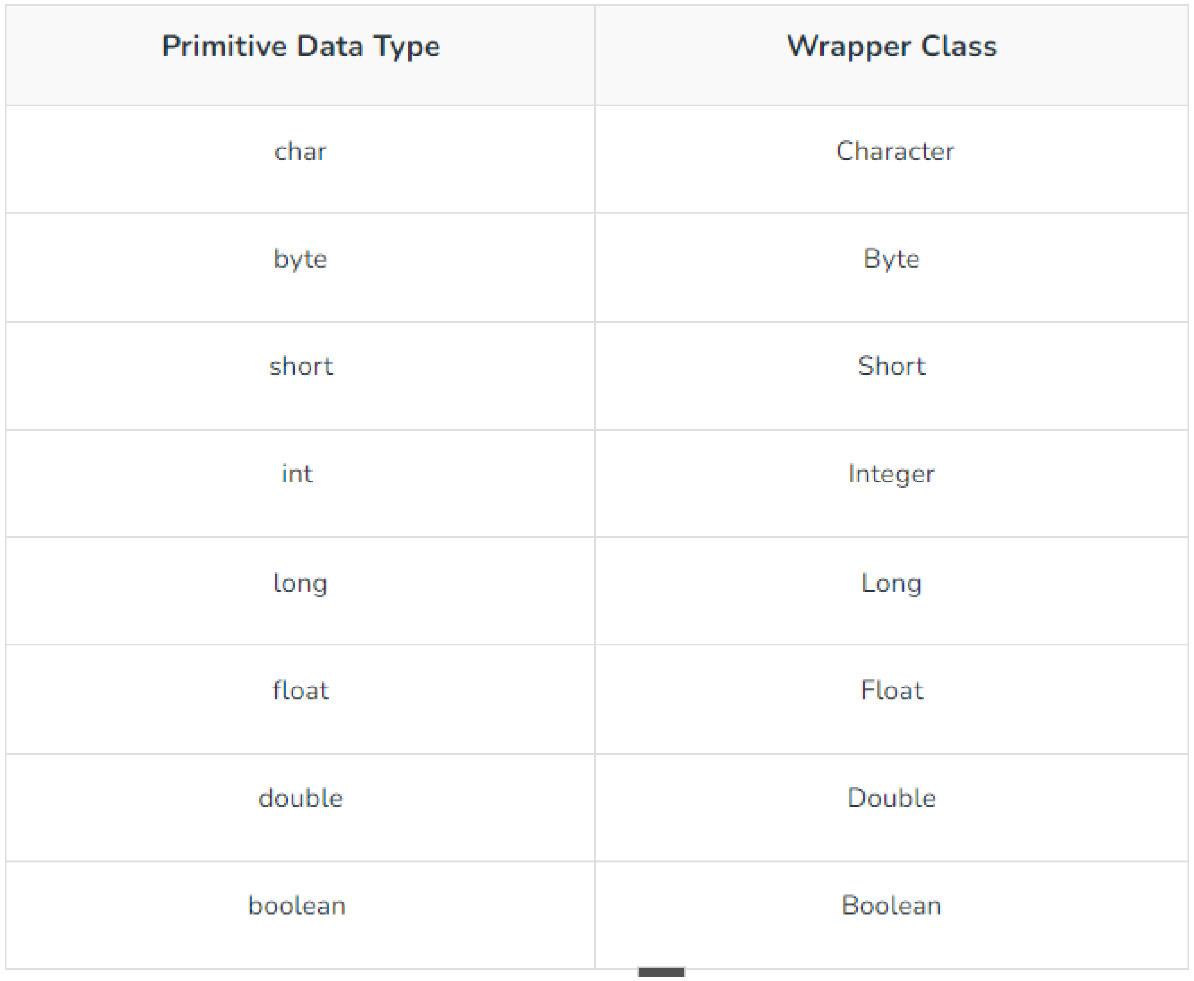
Need of Wrapper Classes

- 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).
- The classes in java.util package handles only objects and hence wrapper classes help in this case also.
- Data structures in the Collection framework, such as ArrayList and **Vector**, store only objects (reference types) and not primitive types.
- An object is needed to support synchronization in multithreading.

Advantages

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

Difference?



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

Autoboxing?

Autoboxing is the automatic conversion that the Java compiler makes between the primitive types and their corresponding object wrapper classes.

For example, converting an int to an Integer, a double to a Double, and so on.

Example

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

public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}}
```

Example 2

```
import java.io.*;
class GFG {
        public static void main(String[] args)
        {
                // Creating an Integer Object
                // with custom value say it be 10
                Integer i = new Integer(10);

                // Unboxing the Object
                int i1 = i;

                System.out.println("Value of i:" + i);
                System.out.println("Value of i1: " + i1);

                // Autoboxing of character
                Character gfg = 'a';

                // Auto-unboxing of Character
                char ch = gfg;

                // Print statements
                System.out.println("Value of ch: " + ch);
                System.out.println(" Value of gfg: " + gfg);
        }
}
```

Unboxing

- The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.
- Since Java 5, we do not need to use the intValue() method of wrapper classes to convert the wrapper type into primitives.

Example

- /Java program to convert object into primitives
- //Unboxing example of Integer to int
- public class WrapperExample2{
- public static void main(String args[]){
- //Converting Integer to int
- Integer a=new Integer(3);
- int i=a.intValue();//converting Integer to int explicitly
- int j=a;//unboxing, now compiler will write a.intValue() internally
- System.out.println(a+" "+i+" "+j);
- }}