unit2

OOPS

Data Types

A **variable** in Java is like a labeled box that stores data in your computer's memory. Each variable has:
Data Type: Specifies the kind of data the variable can hold (e.g., int, String).
Variable Name: A unique identifier you assign to the variable.
Value: The actual data stored in the variable

Data types in Java specify the type of value that a variable can hold. There are two main categories of data types in Java: primitive and non-primitive.

Primitive Data Type

These are predefined by the language and include:

//Whole numbers

byte: 8-bit integer

short: 16-bit integer

int: 32-bit integer

long: 64-bit integer

//Decimal numbers

float: 32-bit floating-point number

double: 64-bit floating-point number

//Characters

char: 16-bit Unicode character

//Boolean

boolean: true or false value

```
byte b=7;
short s= 67;
int a= 1607;
long d= 128901;
double num=3.14;
float num1=7.66f;
char c ='H';
boolean ans= true;
```

Literals in Java is a synthetic representation of boolean, numeric, character, or string data. It is a medium of expressing particular values in the program.

```
int a = 101; // decimal-form literal
int b = 0100;  // octal-form literal
int c = 0xFace;  // Hexa-decimal form literal
int d = 0b1111;  // Binary literal
```

Type Conversion And Type Casting

Type conversion is **automatic** and **implicit**, done by the compiler/interpreter without the programmer's intervention.
Also known as **implicit casting**.

```
int a = 10;
double b = a;  // int is automatically converted to double
```

Type casting is **manual** and **explicit**, done by the programmer when they want to force a conversion.
You write code to convert from one type to another.

```
double a = 10.5;
int b = (int) a;
System.out.println(b);
```

Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main differences between **primitive** and **non-primitive** data types are:

- Primitive types in Java are predefined and built into the language, while non-primitive types are created by the programmer (except for String).
- Non-primitive types can be used to call methods to perform certain operations, whereas primitive types cannot.
- Primitive types start with a lowercase letter (like int), while non-primitive types typically starts with an uppercase letter (like String).
- Primitive types always hold a value, whereas non-primitive types can be null.
- Examples of non-primitive types are Strings, Arrays, Classes etc.

String

The string data type in Java represents a sequence of characters. It is a non-primitive data type, meaning it is a class rather than one of the eight primitive types (byte, short, int, long, float, double, boolean, char). Strings are defined by enclosing characters within double quotes.

String objects are immutable, meaning their values cannot be changed after creation.

```
String name1 ="Drishti";
String name2= new String(original:"Drishti");
```

Java String Methods

Method	Description	Return Type
charAt()	Returns the character at the specified index (position)	char
copyValueOf()	Returns a String that represents the characters of the character array	String
endsWith()	Checks whether a string ends with the specified character(s)	boolean
equals()	Compares two strings. Returns true if the strings are equal, and false if not	boolean
toCharArray()	Converts this string to a new character array	char[]
toLowerCase()	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
toUpperCase()	Converts a string to upper case letters	String

Arrays

An array is a data structure that stores a fixed-size, sequential collection of elements of the same type. These elements are stored in contiguous memory locations, and each element can be directly accessed using its index.
Type[] name= new type[size],

```
int[] intArray = new int[5]; // Creates an integer array of size 5
String[] stringArray = {"apple", "banana", "cherry"}; // Creates and initializes a String array
```

```
int[] numbers = {10, 20, 30, 40, 50};
int firstElement = numbers[0]; // Accesses the first element (10)
int thirdElement = numbers[2]; // Accesses the third element (30)
```

Multidimensional Array

```
// Declaring and initializing a 2D array
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

// Declaring and initializing a 3D array
int[][][] cube = new int[2][3][4];
```

Multidimensional arrays are declared by specifying the data type followed by square brackets for each dimension.
For example, a 2D array of integers is declared as int[][] myArray;.
Initialization can be done at the time of declaration or later using the new keyword.

```
int element = matrix[1][2]; // Accessing the element at row 1, column 2 (value: 6)
```

Wrapper classes

Why java is not a pure object oriented language?

Java is not a fully object-oriented language as it supports primitive data types like int, byte, long, short, etc., which are not objects. Hence these data types like int, float, double, etc., are not object-oriented. That's why Java is not 100% object-oriented.

Wrapper classes in Java serve as object representations of primitive data types, which are not objects. This is essential because many Java functionalities require objects

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.

.

Autoboxing and Unboxing in Java

Autoboxing refers to the conversion of a primitive value into an object of the corresponding wrapper class is called autoboxing. For example, converting int to Integer class.
The Java compiler applies autoboxing when a primitive value is:
•Passed as a parameter to a method that **expects an object** of the corresponding wrapper class.
•Assigned to a variable of the corresponding **wrapper class**.

Unboxing on the other hand refers to converting an object of a wrapper type to its corresponding primitive value. For example conversion of Integer to int.
The Java compiler applies to unbox when an object of a wrapper class is:
•Passed as a parameter to a method that **expects a value** of the corresponding primitive type.
•Assigned to a variable of the corresponding **primitive type**.

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

Java Loops

In Java, there are three types of Loops, which are listed below:

for loop
while loop
do-while loop

Iteration Statements – for

```
for(initializer; condition; incrementer)
{
// statements to keep executing while condition is
true
}
```

Example

```
int i;
int length = 10;
for (i = 0; i < length; i++) {
. . .
// do something to the (up to 9 )
. . .
}
```

Iteration Statements - While

```
while(condition)
   {
   // statements to keep executing while condition is
   true
        ..
        ..
   }
```

Example

```
   //Increment n by 1 until n is greater than 100
   while (n > 100) {
   n = n + 1;
   }
```

Iteration Statements – Do While

```
Do {
  // statements to keep executing while condition is
  true
} while(condition);
```

It will first executes the statement and then evaluates the condition.

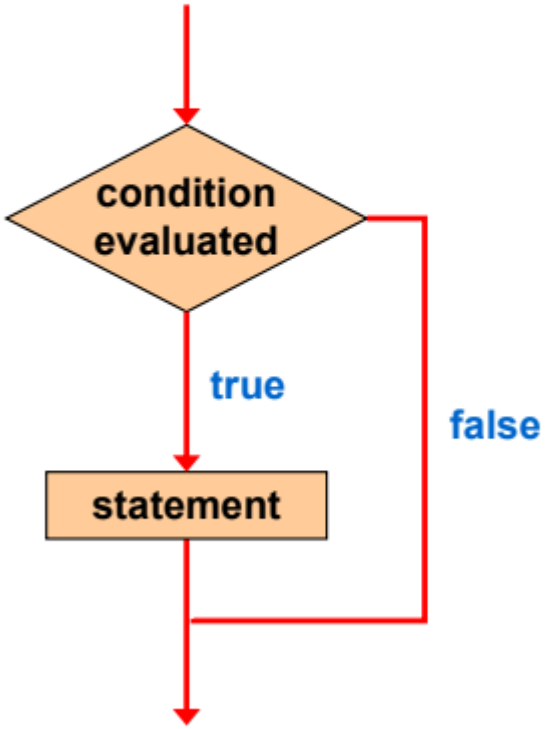
- Example

```
int n = 5;
Do {
   System.out.println(" n = " + n);
   N--;
     } while(n > 0);
```

Control of Flow

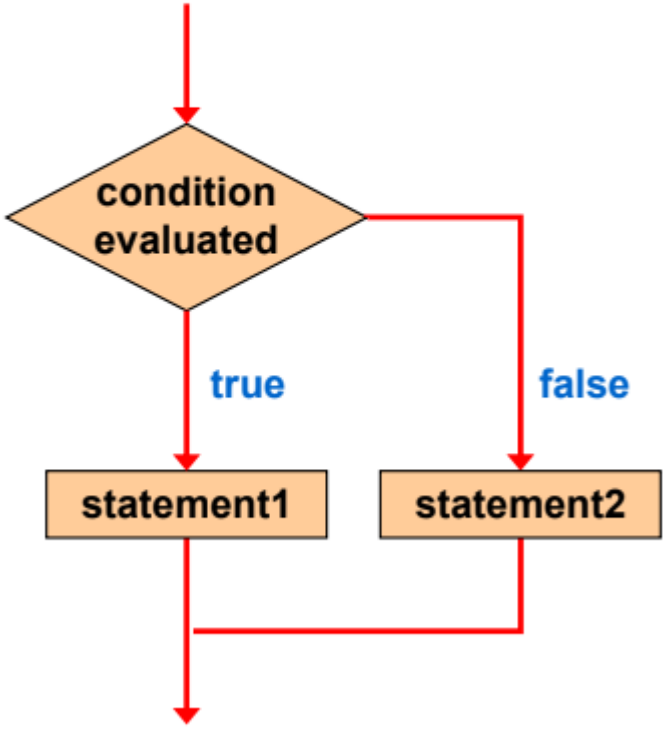
Selection Statements – If

```
if (condition) {
        statement1;
         }
```



Selection Statements – If - else

```
if (condition) {
    statement1;
     }
else {
    statement 2,
  }
```



Selection Statements – Nested Ifs

```
if (num1 < num2)
        if (num1 < num3)
            min = num1;
        else
            min = num3;
    else
        if (num2 < num3)
            min = num2;
        else
            min = num3;
```

```java
for (int i = 1; i <= 10; i++) {

    // Skip number 5 using 'continue'
    if (i == 5) {
        System.out.println(x:"Skipping number 5");
        continue;  // skips rest of loop for i = 5
    }

    // Stop loop at number 8 using 'break'
    if (i == 8) {
        System.out.println(x:"Breaking at number 8");
        break;  // exits the loop
    }
    else{

    // Otherwise, just print the number
    System.out.println("Number: " + i);
    }
}

System.out.println(x:"Loop ended.");
```

Selection Statements – switch

```
switch (expression)
{
    case  value1:
        statement-list1
    case value2:
        statement-list2
    case value3:
        statement-list3
            :
   default:
        statement-list4
}
```

A ***break*** statement causes control to transfer to the end of the switch statement

If a ***break*** statement is not used, the flow of control will continue into the next case

A switch statement can have an optional ***default*** *case*

The default case has no associated value and simply uses the reserved word ***default***

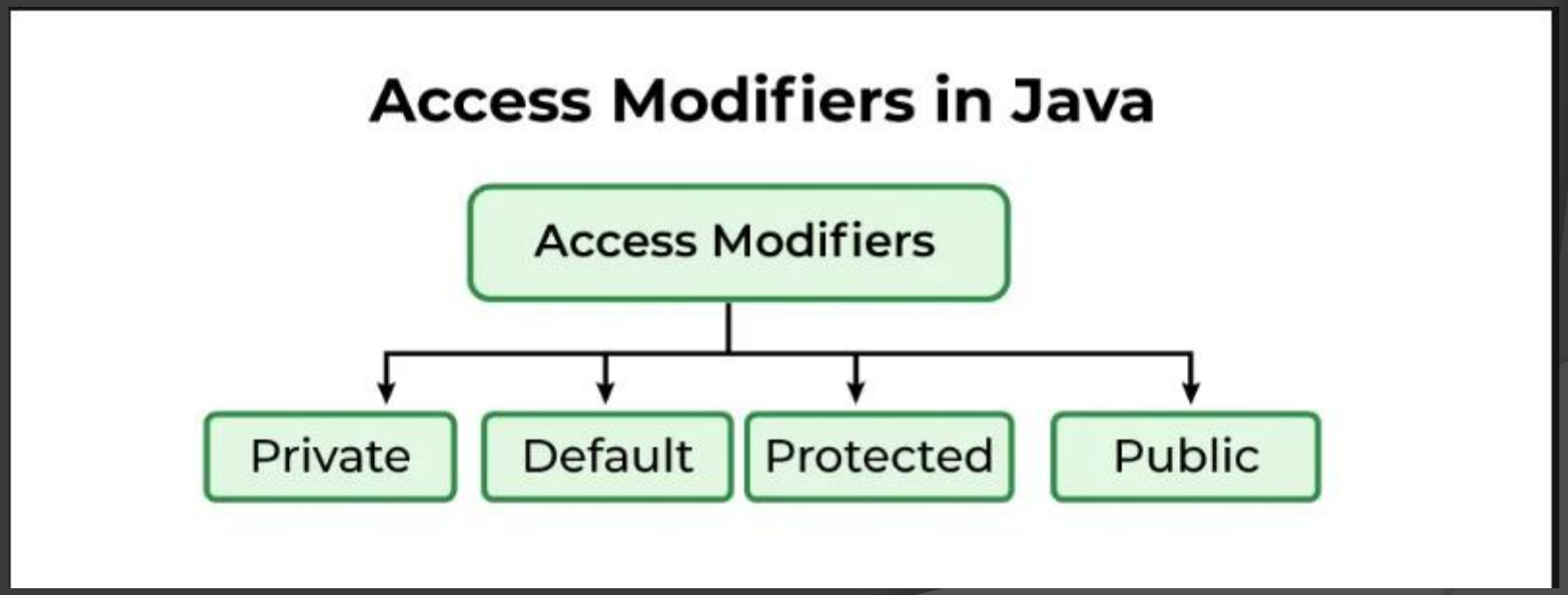
The expression of a `switch` statement must result in an ***integral type***, meaning an **`int`** or a **`char`**

```java
int day = 2;

switch (day) {
    case 1:
        System.out.println("Monday");
        break;

    case 2:
        System.out.println("Tuesday");
        break;

    case 3:
        System.out.println("Wednesday");
        break;

    default:
        System.out.println("Invalid day");
```

Operators	Associativity	Type
++ --	Right to left	Unary postfix
++ -- + - ~ ! (type)	Right to left	Unary prefix
* / %	Left to right	Multiplicative
+ -	Left to right	Additive
<< >> >>>	Left to right	Shift
< <= > >=	Left to right	Relational
== !==	Left to right	Equality
&	Left to right	Boolean Logical AND
^	Left to right	Boolean Logical Exclusive OR
|	Left to right	Boolean Logical Inclusive OR
&&	Left to right	Conditional AND
||	Left to right	Conditional OR
?:	Right to left	Conditional
= += -= *= /= %=	Right to left	Assignment

Class Member Modifier

In Java, **access modifiers** are essential tools that define how the members of a class, like **variables, methods**, and even the **class** itself can be accessed from other parts of our program. They are an important part of building secure and modular code when designing large applications. Understanding **default, private, protected**, and **public** access modifiers is essential for writing efficient and structured Java programs.



•Public:

Members declared public are accessible from anywhere within the project, including other packages.

•Private:

Members declared private are only accessible within the same class.

•Protected:

Members declared protected are accessible within the same package and by subclasses, even if the subclasses are in a different package.

•Default (or package-private):

If no access modifier is specified, the member is accessible only within the same package.

	Default	Private	Protected	Public
Same Class	Yes	Yes	Yes	Yes
Same Package Subclass	Yes	No	Yes	Yes
Same Package Non-Subclass	Yes	No	Yes	Yes
Different Package Subclass	No	No	Yes	Yes
Different Package Non-Subclass	No	No	No	Yes

Encapsulation in Java

Encapsulation is a fundamental principle of Object-Oriented Programming (OOP). In Java, encapsulation refers to the bundling of data (fields or variables) and the methods (functions) that operate on that data into a single unit, typically a class. It also involves restricting direct access to the data by making it private and providing controlled access through public methods, such as getters and setters. This ensures that the internal state of an object is protected from unauthorized or unintended modifications.

Key Points of Encapsulation

- **Data Hiding**: The internal data of a class is hidden from the outside world using the private access modifier.
- **Controlled Access**: Access to the data is provided through public methods (getters to retrieve data and setters to modify data).
- **Bundling**: Data and the methods that manipulate it are bundled together in a class, ensuring they are logically connected.

Why is Encapsulation Important?

Encapsulation provides several benefits:

- **Data Protection**: It prevents unauthorized access to sensitive data by hiding the internal state of an object.
- **Improved Maintainability**: Changes to the internal implementation of a class can be made without affecting the code that uses the class, as long as the public methods remain unchanged.
- **Modularity**: It groups related data and behavior together, making the code easier to understand and manage.
- **Flexibility**: Developers can add logic (e.g., validation) in setters without affecting external code.

```java
public class Student {
    // private data members (encapsulated)
    private String name;
    private int age;

    // public getter method for name
    public String getName() {
        return name;
    }

    // public setter method for name
    public void setName(String newName) {
        name = newName;
    }

    // public getter method for age
    public int getAge() {
        return age;
    }

    // public setter method for age
    public void setAge(int newAge) {
        if (newAge > 0) {
            age = newAge;
        } else {
            System.out.println("Invalid age!");
        }
    }
```

```java
    // Main method to test
    public static void main(String[] args) {
        Student s1 = new Student();
        s1.setName("Drishti");
        s1.setAge(20);

        System.out.println("Name: " + s1.getName());
        System.out.println("Age: " + s1.getAge());

        s1.setAge(-5); // Will show invalid message
    }
}
```

Java Abstraction

Data **abstraction** is the process of hiding certain details and showing only essential information to the user.
Abstraction can be achieved with either **abstract classes** or **interfaces.**

The abstract keyword is a non-access modifier, used for classes and methods:

Abstract class: is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).

Abstract method: can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Abstract classes

- An instance of an abstract class can not be created.
- Constructors are allowed.
- We can have an abstract class without any abstract method.
- There can be a **final method** in abstract class but any abstract method in class(abstract class) can not be declared as final or in simpler terms final method can not be abstract itself as it will yield an error: “Illegal combination of modifiers: abstract and final”
- If a **class** contains at least **one abstract method** then compulsory should declare a class as abstract
- If the **Child class** is unable to provide implementation to all abstract methods of the **Parent class** then we should declare that **Child class as abstract** so that the next level Child class should provide implementation to the remaining abstract method

```java
abstract class Animal {
    // Abstract method
    abstract void makeSound();

    // Concrete method
    void sleep() {
        System.out.println("Sleeping...");
    }
}

class Dog extends Animal {
    void makeSound() {
        System.out.println("Woof!");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        dog.makeSound();  // Output: Woof!
        dog.sleep();      // Output: Sleeping...
    }
}
```

Key features of abstraction:

•Abstraction hides the complex details and shows only essential features.

•Abstract classes may have methods without implementation and must be implemented by subclasses.

•By abstracting functionality, changes in the implementation do not affect the code that depends on the abstraction.

Interfaces

Another way to achieve abstraction in Java, is with interfaces.
An interface is a completely "**abstract class**" that is used to group related methods with empty bodies.

Notes on Interfaces:

•Like **abstract classes**, interfaces **cannot** be used to create objects (in the example above, it is not possible to create an "Animal" object in the MyMainClass)
•Interface methods do not have a body - the body is provided by the "implement" class
•On implementation of an interface, you must override all of its methods
•Interface methods are by default- abstract and public
•Interface attributes are by default- public, static and final.]
•An interface cannot contain a constructor (as it cannot be used to create objects)

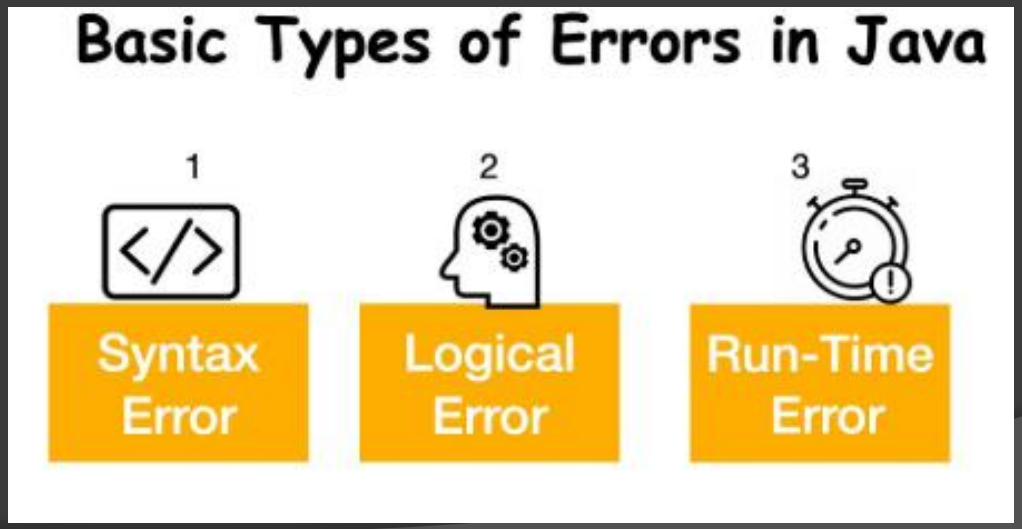
```java
// Define an interface
interface Flyable {
    void fly(); // implicitly public and abstract
}

// Define a class that implements the interface
class Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Bird flies in the sky");
    }
}
```

```java
// Another class implementing the same interface
class Airplane implements Flyable {
    @Override
    public void fly() {
        System.out.println("Airplane flies with engines");
    }
}

public class Main {
    public static void main(String[] args) {
        Flyable f1 = new Bird();
        Flyable f2 = new Airplane();

        f1.fly(); // Output: Bird flies in the sky
        f2.fly(); // Output: Airplane flies with engines
    }
}
```

Exception handling

Exception handling in Java allows developers to manage runtime errors effectively by using mechanisms like **try-catch block**, **finally block**, **throwing Exceptions**, **Custom Exception handling,** etc.

An **Exception** is an unwanted or unexpected event that occurs during the execution of a program (i.e., at **runtime**) and disrupts the normal flow of the program's instructions. It occurs when something unexpected happens, like accessing an invalid index, dividing by zero, or trying to open a file that does not exist.



Try-Catch Block

A try-catch block in Java is a mechanism to handle exception. The try block contains code that might thrown an exception and the catch block is used to handle the exceptions if it occurs.

finally Block

The finally Block is used to execute important code regardless of whether an exception occurs or not.

Note: finally block is always executes after the try-catch block. It is also used for resource cleanup.

```
try {

    // Code that may throw an exception

} catch (ExceptionType e) {

    // Code to handle the exception

}finally{

// cleanup code


}
```

Java exceptions are broadly categorized into **two** main types:

Feature	Checked Exception	Unchecked Exception
Behaviour	Checked exceptions are checked at compile time.	Unchecked exceptions are checked at run time.
Base class	Derived from Exception	Derived from RuntimeException
Cause	External factors like file I/O and database connection cause the checked Exception.	Programming bugs like logical errors cause unchecked Exceptions.
Handling Requirement	Checked exceptions must be handled using a try-catch block or must be declared using the throws keyword	No handling is required
Examples	IOException, SQLException, FileNotFoundException.	NullPointerException, ArrayIndexOutOfBoundsException.

1. **ArithmeticException:** It is thrown when an exceptional condition has occurred in an arithmetic operation.
2. **ArrayIndexOutOfBoundsException:** It is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.
3. **ClassNotFoundException:** This Exception is raised when we try to access a class whose definition is not found
4. **FileNotFoundException:** This Exception is raised when a file is not accessible or does not open.
5. **IOException:** It is thrown when an input-output operation failed or interrupted
6. **InterruptedException:** It is thrown when a thread is waiting, sleeping, or doing some processing, and it is interrupted.
7. **NoSuchFieldException:** It is thrown when a class does not contain the field (or variable) specified
8. **NoSuchMethodException:** It is thrown when accessing a method that is not found.
9. **NullPointerException:** This exception is raised when referring to the members of a null object. Null represents nothing
10. **NumberFormatException:** This exception is raised when a method could not convert a string into a numeric format.
11. **RuntimeException:** This represents an exception that occurs during runtime.
12. **StringIndexOutOfBoundsException:** It is thrown by String class methods to indicate that an index is either negative or greater than the size of the string
13. **IllegalArgumentException :** This exception will throw the error or error statement when the method receives an argument which is not accurately fit to the given relation or condition. It comes under the unchecked exception.
14. **IllegalStateException :** This exception will throw an error or error message when the method is not accessed for the particular operation in the application. It comes under the unchecked exception.

Java final, finally and finalize

In Java, the **final, finally,** and **finalize** keywords play an important role in **exception handling.**
The main difference between final, finally, and finalize is,

final: The “final” is the keyword that can be used for immutability and restrictions in variables, methods, and classes.
finally: The **“**finally block” is used in exception handling to ensure that a certain piece of code is always executed whether an exception occurs or not.
finalize: finalize is a method of the object class, used for cleanup before garbage collection.

```
// Constant value

final int a = 100;
```

```
try {
    // Code that might throw an exception
} catch (ExceptionType e) {
    // Code to handle the exception
} finally {
    // Code that will always execute
}
```

```
protected void finalize throws Throwable{}
```

Java 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 exception. The throw keyword is mainly used to throw custom exceptions.

```
int a = 10;
int b = 0;
if (b == 0) {
    throw new ArithmeticException("/ by zero");
}
int c = a / b; // This line would not execute
```

Java throws

throws is a keyword in Java that is used in the signature of a 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.

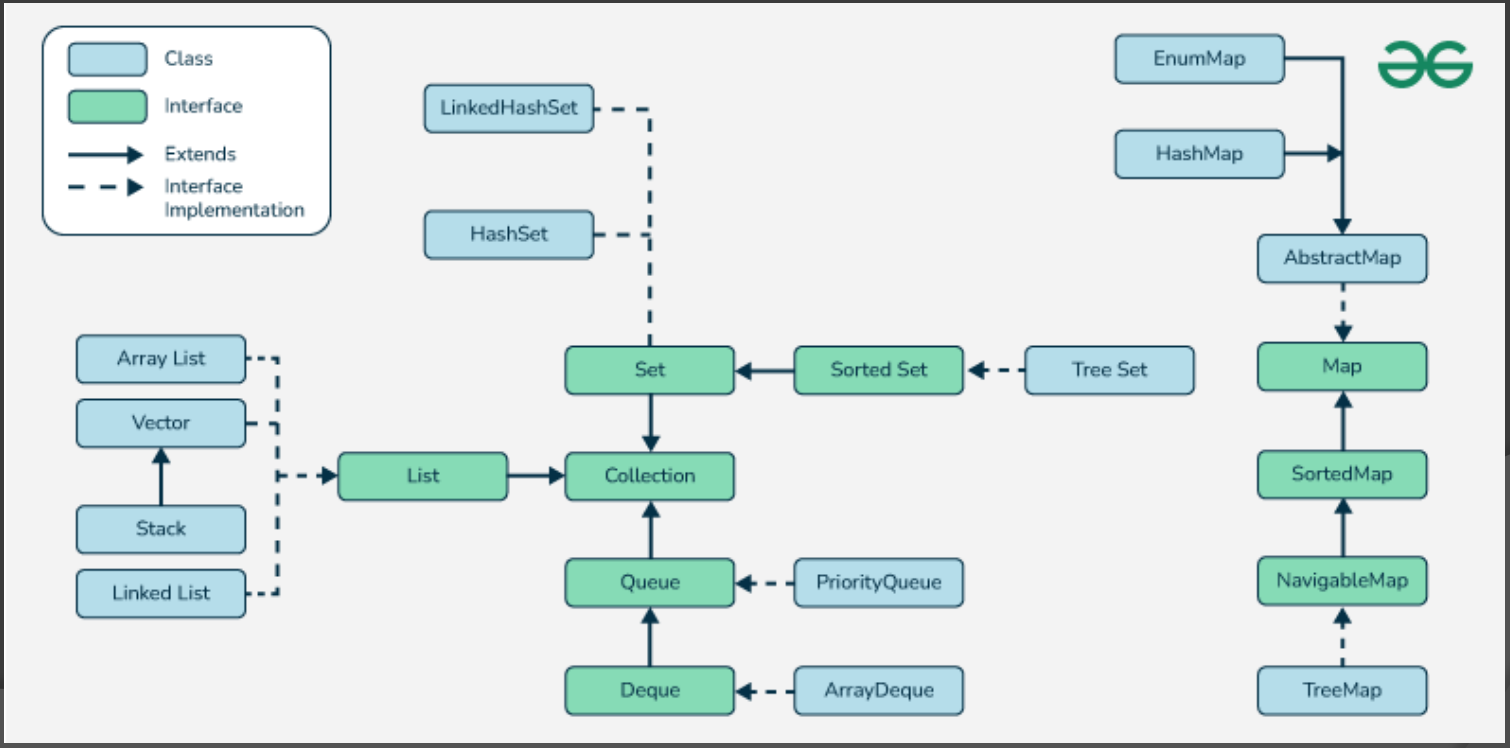
```
import java.io.*;

public class Example {

    // This method declares it might throw IOException
    public static void readFile(String fileName) throws IOException {
        FileReader file = new FileReader(fileName);
        BufferedReader fileInput = new BufferedReader(file);

        // Print the first 3 lines of the file
        for (int counter = 0; counter < 3; counter++)
            System.out.println(fileInput.readLine());

        fileInput.close();
    }

    public static void main(String[] args) {
        try {
            readFile("example.txt");
        } catch (IOException e) {
            System.out.println("An IOException occurred: " + e.getMessage());
        }
    }
}
```

Feature	throw	throws
Definition	It is used to explicitly throw an exception.	It is used to declare that a method might throw one or more exceptions.
Location	It is used inside a method or a block of code.	It is used in the method signature.
Usage	It can throw both checked and unchecked exceptions.	It is only used for checked exceptions. Unchecked exceptions do not require **throws**
Responsibility	The method or block throws the exception.	The method's caller is responsible for handling the exception.
Flow of Execution	Stops the current flow of execution immediately.	It forces the caller to handle the declared exceptions.

Collections in Java

In Java, a separate framework named the *"Collection Framework"* has been defined which holds all the Java Collection Classes and Interface in it.

In Java, the Collection interface (**java.util.Collection**) and Map interface (**java.util.Map**) are the two main "root" interfaces of Java collection classes.



1. List Interface

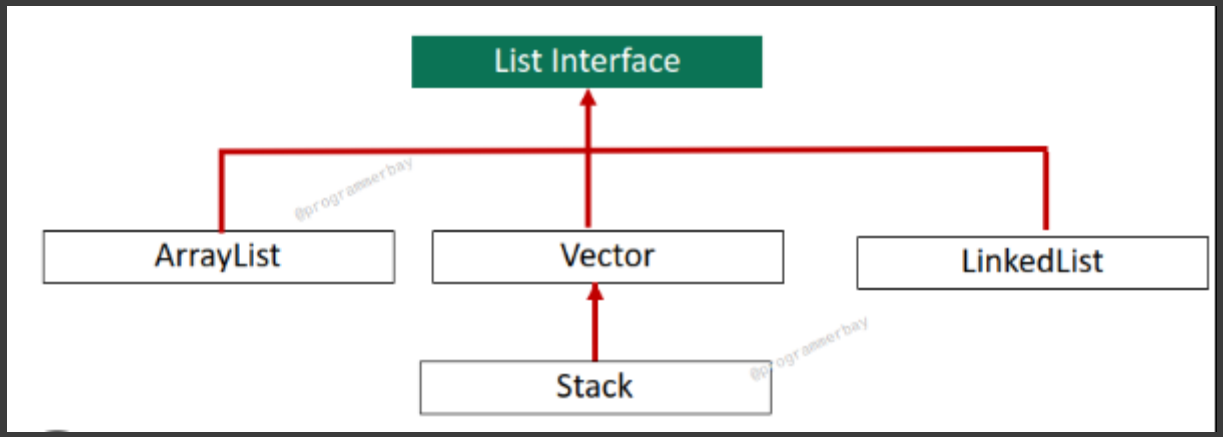
Key Implementations: ArrayList, LinkedList, Vector

Importance:

1. **Ordered** collection – elements are maintained in the order they were inserted.
2. **Allows duplicates**
3. **Indexed access** – elements can be accessed by position.

Use Cases:

1. Maintaining a sequence of elements.
2. Accessing elements by index.
3. When duplicates are meaningful (e.g., list of exam scores, tasks, etc.).



Java ArrayList

The ArrayList class is a resizable array, which can be found in the java.util package.

The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want. The syntax is also slightly different.

```java
import java.util.ArrayList;
```

```java
ArrayList<Type> listName = new ArrayList<>();
```

```java
ArrayList<String> fruits = new ArrayList<>();
ArrayList<Integer> numbers = new ArrayList<>();
```

- **Add elements**

```java
fruits.add("Apple");
fruits.add("Banana");
```

- **Get an element**

```java
String fruit = fruits.get(0);  // "Apple"
```

- **Set an element**

```java
fruits.set(1, "Mango");
```

- **Remove an element**

```java
fruits.remove("Apple");
fruits.remove(0);  // removes first element
```

- **Size of the list**

```java
int size = fruits.size();
```

Java LinkedList

```
import java.util.LinkedList;

public class Main {
  public static void main(String[] args) {
    LinkedList<String> cars = new LinkedList<String>();
    cars.add("Volvo");
    cars.add("BMW");
    cars.add("Ford");
    cars.add("Mazda");
    System.out.println(cars);
  }
}
```

The LinkedList class has all of the same methods as the ArrayList class because they both implement the List interface. This means that you can add items, change items, remove items and clear the list in the same way.

Java Stack

In Java, a **Stack** is a class that represents a **last-in, first-out (LIFO)** collection of objects. It extends **Vector** and implements the **Deque** interface. The Stack class allows you to push and pop elements in a LIFO order.

Key Features of Stack
Push: Add an element to the top of the stack.
Pop: Remove the element from the top of the stack.
Peek: View the element at the top of the stack without removing it.
Search: Find the position of an element in the stack.

Basic Stack Operations
push(E item): Adds an element to the top of the stack.
pop(): Removes and returns the element at the top of the stack.
peek(): Returns the element at the top of the stack without removing it.
isEmpty(): Checks if the stack is empty.
search(Object o): Returns the 1-based position of the object from the top of the stack.

```java
import java.util.Stack;

public class StackExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();
        final int MAX_SIZE = 3;

        // Push with overflow check
        for (int i = 1; i <= 4; i++) {
            if (stack.size() >= MAX_SIZE) {
                System.out.println("Overflow: Cannot push " + i);
            } else {
                stack.push(i);
                System.out.println("Pushed: " + i);
            }
        }

        // Pop with underflow check
        for (int i = 0; i < 4; i++) {
            if (stack.isEmpty()) {
                System.out.println("Underflow: Stack is empty");
            } else {
                System.out.println("Popped: " + stack.pop());
            }
        }
    }
}
```

2. Set Interface

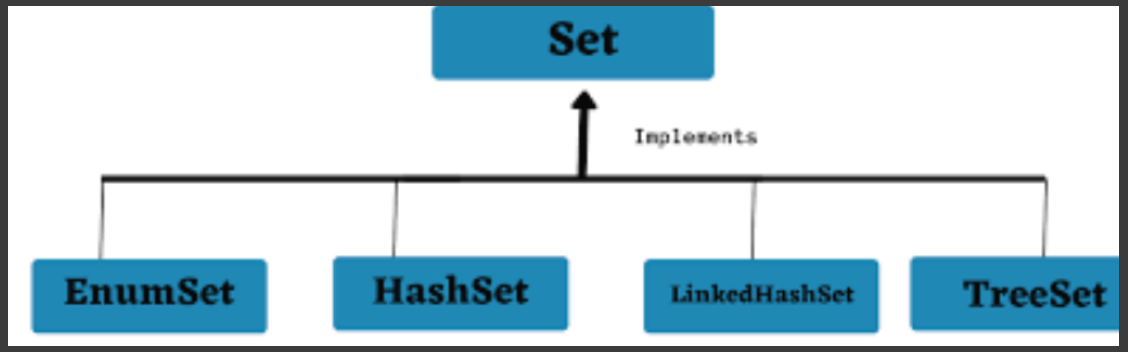
Key Implementations: HashSet, LinkedHashSet, TreeSet

Importance:

- **No duplicates allowed**
- Often **unordered** (HashSet) or **sorted** (TreeSet)
- Used for **uniqueness checks**

Use Cases:

- Storing unique items like usernames, IDs, email addresses.
- Filtering duplicates from a collection.



```java
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        // Create a HashSet
        HashSet<String> cars = new HashSet<String>();

        // Add elements
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("BMW");  // Duplicate, won't be added

        // Print the HashSet
        System.out.println("Cars: " + cars);

        // Check if an item exists
        if (cars.contains("Ford")) {
            System.out.println("Ford is in the set.");
        }

        // Remove an element
        cars.remove("Volvo");

        // Size of the HashSet
        System.out.println("Set size: " + cars.size());
```

3. Map Interface

Key Implementations: HashMap, LinkedHashMap, TreeMap

Importance:

- Stores **key-value pairs**
- Keys must be **unique**, values can be duplicated.
- Efficient lookup, insert, and delete by key.

Use Cases:

- Representing a dictionary or lookup table.
- Associating values with keys (e.g., studentId → studentName).

```java
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        // Create a HashMap
        HashMap<String, String> capitalCities = new HashMap<String, String>();

        // Add key-value pairs
        capitalCities.put("India", "New Delhi");
        capitalCities.put("USA", "Washington D.C.");
        capitalCities.put("UK", "London");
        capitalCities.put("Australia", "Canberra");

        // Print the whole map
        System.out.println("Capital Cities: " + capitalCities);

        // Access a value using a key
        System.out.println("Capital of India: " + capitalCities.get("India"));

        // Remove a key-value pair
        capitalCities.remove("UK");
```

REPEATED QUESTIONS

OR

What are the differences between the final, finally, and finalize keywords in Java? Also, design a Java program that demonstrates the use of the final keyword for variables, methods, and classes.

Difference between throws and throw

How is multiple inheritance possible in java explain

Explain different types of exception. What types of exceptions get caught during compile time explain with example?

What role does the Sandbox model play in improving security and flexibility within java applications?

Differentiate between method overloading and method overriding with suitable example.

```java
public class FinalVariableExample {
    public static void main(String[] args) {
        final int x = 10;
        System.out.println("x = " + x);
        // x = 20; // Error: cannot assign a value to final variable 'x'
    }
}
```

```java
class Parent {
    public final void showMessage() {
        System.out.println("This is a final method.");
    }
}

class Child extends Parent {
    // public void showMessage() {  // Error: cannot override final method
    //     System.out.println("Trying to override.");
    // }
}
```

```java
final class Vehicle {
    public void start() {
        System.out.println("Vehicle started.");
    }
}

// class Car extends Vehicle {  // Error: cannot inherit from final class
// }

public class FinalClassExample {
    public static void main(String[] args) {
        Vehicle v = new Vehicle();
        v.start();
    }
}
```

UNIT-I

Q2 a) Explain Sandbox model architecture. Also explain its significance with respect to the web based Java applications. **(8)**

(i) Write a single java program for the implementation of following object oriented concepts: **(7)**

(a) Encapsulation
(b) Inheritance
(c) Polymorphism
(d) Abstraction

OR

Q3 (a) Explain all the steps followed for execution of a Java program. **(7)**

(a) Write a Java program for the printing the fibonacci series. **(4+4)**

(b) Differentiate between procedural programming and object oriented programming.

```java
public class FibonacciExample {
    public static void main(String[] args) {
        int n = 10; // number of terms in the Fibonacci series
        int first = 0, second = 1;

        System.out.println("Fibonacci Series up to " + n + " terms:");

        for (int i = 1; i <= n; i++) {
            System.out.print(first + " ");

            // compute the next term
            int next = first + second;
            first = second;
            second = next;
        }
    }
}
```

UNIT-II

Q4 (i) Define wrapper classes. Write a java program to perform autoboxing for conversion of all 8 primitive data types to corresponding objects. (7)

(ii) Why exception handling is required? Explain the different Java keywords used for exception handling. (8)

OR

Q5 (i) Differentiate between collections and collection framework. (4+4)

(ii) Define the scenarios where we can use ArrayList instead of an Array.

(iii) Write a Java program to perform push and pop operations on stack with the following conditions: (7)

(a) The maximum capacity of stack is 20.

(b) While performing push operation, overflow condition should be checked

(c) While performing pop operation, underflow condition should be checked.

Definition of Wrapper Classes: Wrapper classes in Java provide a way to use primitive data types (e.g., int, char, double) as objects. Each primitive type has a corresponding wrapper class in the java.lang package. These classes are used to encapsulate primitive values in objects, enabling them to be used in situations where objects are required (e.g., in collections like ArrayList).

The eight primitive types and their corresponding wrapper classes are:
byte → Byte
short → Short
int → Integer
long → Long
float → Float
double → Double
char → Character
boolean → Boolean
Autoboxing: Autoboxing is the automatic conversion of a primitive type to its corresponding wrapper class object by the Java compiler. For example, converting an int to an Integer.

```java
public class AutoboxingExample {
    public static void main(String[] args) {
        // Primitive variables
        byte b = 10;
        short s = 100;
        int i = 1000;
        long l = 10000L;
        float f = 3.14f;
        double d = 99.99;
        char c = 'A';
        boolean bool = true;

        // Autoboxing: Converting primitives to wrapper objects
        Byte byteObj = b;
        Short shortObj = s;
        Integer intObj = i;
        Long longObj = l;
        Float floatObj = f;
        Double doubleObj = d;
        Character charObj = c;
        Boolean booleanObj = bool;

        // Displaying wrapper objects
        System.out.println("Byte object: " + byteObj);
        System.out.println("Short object: " + shortObj);
        System.out.println("Integer object: " + intObj);
        System.out.println("Long object: " + longObj);
        System.out.println("Float object: " + floatObj);
        System.out.println("Double object: " + doubleObj);
        System.out.println("Character object: " + charObj);
        System.out.println("Boolean object: " + booleanObj);
    }
}
```

Why Exception Handling is Required: Exception handling is essential in Java to manage runtime errors (exceptions) that can disrupt the normal flow of a program.

It ensures:

Robustness: Prevents the program from crashing by handling errors gracefully.

Error Recovery: Allows the program to recover from errors and continue execution.

Debugging: Provides detailed error information (e.g., stack trace) for debugging.

User Experience: Displays meaningful error messages instead of abrupt termination.

Resource Management: Ensures resources (e.g., files, database connections) are properly closed even if an error occurs.

Without exception handling, uncaught exceptions can cause program termination, data loss, or unpredictable behavior.

Aspect	Collections	Collection Framework
Definition	Refers to a group of objects stored together (e.g., lists, sets).	A unified architecture in Java for representing and manipulating collections.
Scope	General term for any group of objects (e.g., arrays, ArrayList).	A specific set of interfaces, classes, and algorithms in `java.util`.
Structure	Can be implemented in various ways (e.g., custom classes, arrays).	Standardized hierarchy of interfaces (e.g., List, Set) and classes (e.g., ArrayList, HashSet).
Features	May or may not provide built-in methods for manipulation.	Provides methods for searching, sorting, iteration, etc.
Example	An array or a custom list implementation.	`ArrayList`, `HashMap`, `TreeSet`, etc., from `java.util`.

Q2 Discuss the main features of object-oriented programming. Explain how each feature contributes to code reusability, maintainability, and scalability. Provide examples in Java to illustrate each feature. **(12.5)**

OR

Q3 Describe the life cycle of a Java program starting from source code to execution within the JVM. Discuss how each stage ensures the correct and efficient running of Java applications. **(12.5)**

Q4 Explain the role and significance of the Set, List, and Map interfaces in the Java Collections Framework. Write a Java program that demonstrates the use of Set, List, and Map interfaces. Use *Iterator* to iterate over the elements of a HashSet, an ArrayList, and a HashMap. Add and remove elements, and show how iteration is performed. **(12.5)**

Stage	Role in Correctness	Role in Efficiency
Source Code	Syntax adherence	Optimized algorithms
Compilation	Type safety, error detection	Bytecode optimization
Class Loading	Security and verification	Class caching
Bytecode Verification	Prevents illegal operations	Avoids runtime errors
Interpretation/JIT	Accurate execution	Native code optimization
Execution	Runtime checks, exception handling	Memory management, multithreading
Termination	Resource cleanup	Minimizes system overhead

Iterator

- An **Iterator** is an object that implements the java.util.Iterator interface. It allows you to: Traverse elements in a collection one by one.
- Check if there are more elements to process.
- Remove elements from the collection during iteration (safely).
- It abstracts the traversal process, making it independent of the collection's internal implementation (e.g., whether it's an array-based ArrayList or a hash table-based HashSet).

Syntax of Iterator

To use an `Iterator`, you typically:

1. Obtain an `Iterator` object from a collection using the `iterator()` method.
2. Use a `while` loop with `hasNext()` and `next()` to traverse the collection.

General Syntax:

java

```java
import java.util.Iterator;
import java.util.CollectionType; // e.g., ArrayList, HashSet

CollectionType<Type> collection = new CollectionType<>();
// Populate the collection
Iterator<Type> iterator = collection.iterator();

while (iterator.hasNext()) {
    Type element = iterator.next();
    // Process element
}
```

The `Iterator` interface defines three main methods:

1. **`boolean hasNext()`**:
   - Returns `true` if there are more elements to iterate over, `false` otherwise.
2. **`E next()`**:
   - Returns the next element in the collection.
   - Throws `NoSuchElementException` if there are no more elements.
3. **`void remove()`** (optional):
   - Removes the last element returned by `next()` from the collection.
   - Throws `IllegalStateException` if called before `next()` or after another `remove()` without a `next()`.

```java
import java.util.*;

public class CollectionsDemo {
    public static void main(String[] args) {
        // 1. Set: HashSet
        Set<String> set = new HashSet<>();
        // Adding elements
        set.add("Apple");
        set.add("Banana");
        set.add("Orange");
        set.add("Apple"); // Duplicate, ignored
        System.out.println("HashSet: " + set);

        // Removing an element
        set.remove("Banana");
        System.out.println("After removing Banana: " + set);

        // Iterating using Iterator
        System.out.println("Iterating HashSet:");
        Iterator<String> setIterator = set.iterator();
        while (setIterator.hasNext()) {
            System.out.println(setIterator.next());
        }

        System.out.println("\n--------------------\n");
```