**Language Fundamentals**

**→** In Java, an identifier is a name given to a variable, method, class, interface, or package.

**→** The only allowed characters for identifiers are all alphanumeric characters([A-Z], [a-z], [0-9]), '$'(dollar sign) and '\_'

(underscore). Identifiers can't start with digits.

**→** Java, like most programming languages, is case sensitive.

**→** There is no limit on the length of the identifier, but it is advisable to use an optimum length of 4 – 15 letters only.

**→** Keywords and reserved words in Java, such as int, class, or static, cannot be used as identifiers.

**→** In Java, you **can use predefined class and interface names as identifiers**, but it is not recommended as it reduces code

readability and can lead to confusion (int String = 10; // Valid but discouraged)

**→** Reserved Words: Includes all words that are reserved by Java, which cannot be used as identifiers. These consist of

keywords and **reserved literals (true, false, null).**

**→** Java has **71 reserved** words as of now. These include **keywords (51)**, **reserved literals (3)** and **ContextualKeyword (17)**. The keywords (const and goto) are reserved, they are not currently used.

**→** **Keywords have specific functionalities in Java**, while reserved literals are just reserved for special values and cannot be

used as identifiers.

**→** In Java, ContextualKeyword refers to a word that normally has a specific meaning (like a reserved keyword) but can be used as an identifier under certain conditions.

**→** **In Java, specifying a method's return type is mandatory.** If a method does not return any value, the return type must be

explicitly declared as void.

**→** **'null'** is the default value for all uninitialized object reference variables in Java.

**→ An enum** **in Java is a special data type used to define a collection of constants.** It provides a way to group related values

under a single type and ensures type safety.

**Ex:** public enum Day {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;

}

**→** The destruction of useless or unreferenced objects is the responsibility of the Garbage Collector in Java, which automatically manages memory and reclaims unused objects to free up space.

**→** Java is a strongly typed programming language. This means that every variable in Java must be declared with a data type, and the data type cannot be changed once it's declared.

**→** Java is not considered a pure object-oriented language because it supports primitive data types that are not objects, and it allows the use of static methods and variables.

**→ Java has 8 primitive data types:** byte, short, int, long, float, double, boolean, and char. In Java, signed datatypes refer to the integer types that can represent both positive and negative values.

**→** In the context of binary representation, **a sign bit of 0 indicates that the number is positive**, while a sign bit of 1 signifies that the number is negative.

**→** In Java, the primitive data types, their sizes in bits, and their ranges are as follows:

byte: 8 bits (1 byte), Range: -128 to 127

short: 16 bits (2 bytes), Range: -32,768 to 32,767

int: 32 bits (4 bytes), Range: -2,147,483,648 to 2,147,483,647

long: 64 bits (8 bytes)

float: 32 bits (4 bytes)

double: 64 bits (8 bytes)

char: 16 bits (2 bytes) (The range of char values is from 0 to 65,535 inclusive)

boolean: Size is not precisely defined it can hold one of two values: true or false

**→** 'byte' is the best choice for handling data streams, whether from files or networks, as both file-supported formats and network-supported formats are based on bytes. It is efficient for reading and writing raw binary data.

**→** Float can be used for cases that require memory efficiency. Double is preferred for operations that require extreme accuracy, such as scientific computations and financial applications.

**→** Java is a Unicode-based language, which means it uses the Unicode standard to consistently and uniquely encode characters from various written languages.

**→** In Java, wrapper classes are objects that encapsulate primitive data types, allowing them to be treated as objects. The wrapper classes for each primitive type are as follows:

byte → Byte short → Short int → Integer long → Long

float → Float double → Double char → Character boolean → Boolean

**→** Here are the default values of all primitive data types in Java: byte: 0, short: 0, int: 0, long: 0L, float: 0.0f, double: 0.0d, char: \u0000 (ASCII code 0) (null character), boolean: false

**→** In Java**, literals are constant values** that represent fixed data directly in the code. They can be assigned to variables or used in expressions.

**→** In Java, integral literals can be specified in four ways for integral data types:

**Decimal** (Base 10) (0-9): **Example:** int num = 10;

**Octal** (Base 8) (0-7): Must start with 0. **Example:** int num = 012; // Equivalent to 10 in decimal

**Hexadecimal** (Base 16) (0-9, A-F/a-f):Must start with 0x or 0X. **Example:** int num = 0xA; // Equivalent to 10 in decimal

**Binary** (Base 2) (0 & 1):Must start with 0b or 0B. **Example:** int num = 0b1010; // Equivalent to 10 in decimal

**→** By default, every integral literal in Java is of type int. To explicitly change it to long, we can use the **suffix L or l.**

**→** By default, every floating-point literal in Java is of type double. To explicitly change it to float, we can use the **suffix f or F.**

**→** Floating-point literals can only be specified in decimal format in Java. Specifying them in hexadecimal or octal format results in a compile-time error.

**→** You can assign integral literals to floating-point variables, and those integral literals can be specified in decimal, octal, or hexadecimal formats.

float num1 = 012; // Octal (equivalent to 10 in decimal)

float num2 = 0xA; // Hexadecimal (equivalent to 10 in decimal)

**→** Floating point literals can be written in either standard or scientific notation. For example, you can use a syntax like **1.23e4**, which represents 12300.0. Both float and double types can utilize this notation, and they can also be expressed as decimal fractions.

**→** In Java, a char literal is represented by a single character enclosed within single quotes**. Ex: char ch = 'a';**

**→** You can specify character literal as integral literal which represents UNICODE value of that character. The integral literal can be specified either in decimal or octal or hexadecimal forms. The allowed range is 0 to 65535. You can represent a character literal in Unicode by using the syntax \u followed by four hexadecimal digits. Ex: The Unicode of 'a' in hexadecimal format is \u0061.

**→** Every escape character is a valid character literal in Java. Escape characters like \n (newline) and \t are interpreted by the

Java compiler as single-character literals that convey specific meanings or functions. In Java, escape characters are special

sequences that begin with a backslash (\). **Ex:** **\\** : Backslash**, \'** : Single quote**, \"** : Double quote**, \n** : Newline, **\r** : Carriage

return**, \t** : Tab**, \b** : Backspace, **\f** : Form feed

**→** In Java, any sequence of characters enclosed within double quotes is referred to as a string.

**→** In Java SE 7 and later, you can use **underscores (\_)** between digits in numeric literals to enhance readability. Any number of underscore characters ( \_ ) can appear anywhere between digits. This feature allows underscores to appear anywhere within the number, but not at the beginning or end, or next to a decimal point.

**→** You can assign a long value to a float variable in Java because float has a larger range, even though they represent values differently in memory. However, this may result in precision loss due to the difference in internal representation.

**→** An array is an indexed collection of a fixed number of homogeneous data elements, meaning all elements in the array must be of the same data type. **Example:** int[] numbers = {1, 2, 3, 4, 5}; // Homogeneous data elements of type int

**(Note:** The size of the array is fixed once it is declared**)**

**→** To declare an array in Java, syntax: dataType[] arrayName; For example, to declare an array of integers: int[] numbers; You can also initialize the array at the time of declaration, like this: int[] numbers = {1, 2, 3, 4, 5};

**→** At the time of array declaration, you cannot specify its size, the size must be specified during initialization or when creating the array object. **Ex:** int[] arr; // Declaration (size not specified) arr = new int[5]; // Initialization with size

**→** A simple example of a 2D array declaration in Java: **int[][] matrix;** (Declaration of a 2D array)

matrix = new int[3][3]; // Initialization with 3 rows and 3 columns

**→** If we specify the dimension before the variable in a declaration, it is only allowed for the first variable in the declaration. Attempting to specify the dimension before other variables in the same declaration will result in a compile-time error.

**Ex:** int[] arr1, arr2; // Valid: Both are 1D arrays

int[] arr3[], arr4; // Valid: arr3 is a 2D array, arr4 is a 1D array

int[] arr5, []arr6; // Error: Dimension specified before the second variable

**→** Here’s a simple example of a 3D array declaration in Java: **int[][][] array;** (Declaration of a 3D array)

array = new int[2][3][4]; // Initialization with 2x3x4 dimensions

**→** Every array in Java is an object, which is why it can be created using the new keyword. The new keyword allocates memory for the array dynamically. int[] arr = new int[5]; // Array object

**→** For every array type in Java, a corresponding class is available. These classes are part of the Java language but are not accessible at the programmer level.

**Ex:** int[] arr = new int[5];

System.out.println(arr.getClass().getName()); // Output: [I ([I indicates a 1D integer array)

**→** In Java, **it is legal to create an array with size 0**. The array will be empty, but the reference will still point to a valid array object. Example:

int[] arr = new int[0]; // Valid, but the array has no elements

System.out.println(arr.length); // Output: 0

**(Note:** Accessing any index in a zero-sized array (e.g., arr[0]) will throw an **ArrayIndexOutOfBoundsException)**

**→** If you specify a negative size while creating an array, Java will throw a runtime exception called **NegativeArraySizeException. Ex:** int[] arr = new int[-5]; // Throws NegativeArraySizeException

**(Note:** This exception occurs at runtime, not at compile time**)**

**→** To specify an array size in Java, only the **integer types byte, short, char, or int are allowed.** Using any other data type (long, float, double, boolean, or objects) will result in a compile-time error.

**Ex:** byte b = 5;

int[] arr1 = new int[b]; // Valid

long l = 10;

int[] arr2 = new int[l]; // Compile-time error

**→** The maximum array size allowed in Java is 2147483647 (Integer.MAX\_VALUE), but attempting to allocate such a large array may result in an **OutOfMemoryError due to insufficient heap memory.**

**Ex:** int[] arr = new int[2147483647]; // May throw OutOfMemoryError

(Note: The actual limit depends on the available heap memory and JVM configuration)

**→** In Java, multidimensional arrays are not implemented using a matrix representation, instead, they are implemented as arrays of arrays, making them more memory efficient. This allows jagged arrays, where different rows can have different lengths, reducing wasted memory. (Unlike languages like C/C++, Java does not allocate a fixed rectangular memory block for multidimensional arrays)

**Ex:** int[][] arr = {

{1, 2, 3},

{4, 5},

{6, 7, 8, 9}

}; // Jagged array: each row has a different length

**→** In Java, when an array is created, every element is automatically initialized with a default value based on its type.

**Ex:** int[] arr = new int[5];

System.out.println(arr[0]); // Output: 0 (Key Point: These default values ensure that no uninitialized memory is

accessed in Java)

**→** When you print any reference variable in Java, internally the toString() method is called. By default, toString() in Object class is designed to return: **ClassName@HashCode\_in\_Hexadecimal**

(**Key Point:** You can override toString() in your class to provide a custom string representation)

**→** Java program that declares a 2D array by specifying only the base array size, then prints the array and its elements:

public class Main {

public static void main(String[] args) {

int[][] x = new int[3][];

System.out.println(x); // Output: [[I@<hashcode>

System.out.println(x[0]); // Output: null (since no row is initialized)

System.out.println(x[0][0]); // Throws NullPointerException

}

} (Key Point: If you only specify the base size in a 2D array, the rows must be initialized separately before accessing elements)

**→** In Java, performing any operation on a null reference will result in a **NullPointerException.**

**→** Accessing an array with an out-of-bounds index, either positive or negative, will result in an

**ArrayIndexOutOfBoundsException.**

**→ SCP** (String Constant Pool) is a special memory area **inside the heap** where string literals are stored to avoid duplication and save memory. Stores only string literals, not new String("example") objects. **(Key Point: Strings in SCP are immutable and automatically managed by the JVM)**

**Ex:** String s1 = "Hello";

String s2 = "Hello";

System.out.println(s1 == s2); // true (same SCP reference)

String s3 = new String("Hello");

System.out.println(s1 == s3); // false (different memory)

**→** You cannot initialize an array like this: int[] x;

x = {10, 20, 30}; // Compilation Error (This results in a "cannot find symbol" error)

Correct Ways to Initialize an Array:

int[] x = {10, 20, 30}; // Allowed

int[] x;

x = new int[]{10, 20, 30}; // Allowed ( Array initializer ({}) must be used during declaration, or you must use new int[] explicitly )

**→ 'length'** is a **final variable** in Java, applicable to arrays, and it represents the size of the array. Since length is final, it cannot be modified after array creation.

**Ex:** int[] arr = {10, 20, 30, 40};

System.out.println(arr.length); // Output: 4

arr.length = 10; // Compilation Error (Cannot assign a value to final variable 'length')

**(Key Point:** length is a variable (not a method) for arrays, but **length()** is a method for strings**)**

**→** In a multi-dimensional array, the length variable only represents the size of the base array (i.e., the number of rows in a 2D array), not the total number of elements.

**Ex:** int[][] arr = {

{1, 2, 3},

{4, 5},

{6, 7, 8, 9}

};

System.out.println(arr.length); // Output: 3 (Number of rows)

System.out.println(arr[0].length); // Output: 3 (Columns in the first row)

**→** An anonymous array is an array without a reference variable, created for one-time use. (Used when an array is required only once (e.g., method arguments))

**Ex:** public static void main(String[] args) {

sum(new int[]{10, 20, 30, 40}); // Anonymous array

}

**→** When creating an anonymous array in Java, you cannot specify the size of the array. If you attempt to do so, you will encounter a compile-time error.

**→** In a primitive array, you can assign values of smaller data types, and they will be implicitly promoted to the array's declared data type.

**Ex:** int[] arr = {10, 'A', (byte) 5, (short) 20};

System.out.println(arr[1]); // Output: 65 ('A' promoted to int)

Implicit Promotion Allowed: **byte → short → int → long → float → double char → int**

(Key Point: The assigned values should be compatible with the array type to avoid compilation errors)

**→** In object type arrays, you can assign objects of the declared type or its subclasses as elements.

**Ex:** class Parent {}

class Child extends Parent {}

public class Test {

public static void main(String[] args) {

Parent[] arr = new Parent[3];

arr[0] = new Parent(); // Allowed

arr[1] = new Child(); // Allowed (Child is a subclass of Parent)

arr[2] = new String("Hello"); // CE: Incompatible types

}

}

**Allowed:** Same class type or subclass objects  **Not Allowed:** Unrelated class types

(Object arrays follow polymorphism, allowing upcasting (assigning child objects to parent type arrays))

**→** In interface type arrays, only objects of classes that implement the interface can be assigned as elements. If you try to assign an object of a class that does not implement the interface, you'll get a Compilation Error (CE). (Interface arrays follow polymorphism, allowing only objects of implementing classes)

**→** In abstract class type arrays, only objects of concrete (non-abstract) subclasses of that abstract class are allowed as elements. If you try to assign an object of a class that does not extend the abstract class, you'll get a Compilation Error (CE). (Abstract class arrays follow polymorphism, allowing only instances of concrete subclasses)

**→** Element promotions are applicable at the individual element level but not at the array level. You cannot assign a char[] array to an int[] array, even though individual char values can be promoted to int**. Primitive array types must match exactly; implicit type promotion does not apply to entire arrays.**

**Ex:** char c = 'A';

int i = c; // Allowed (char → int promotion)

char[] charArray = {'A', 'B', 'C'};

int[] intArray = charArray; // CE: Incompatible types

**→ In object type arrays, an array of a child class can be assigned to a parent class array.** This is because arrays follow polymorphism in Java. However, attempting to store an unrelated object type in the parent class reference will lead to a ArrayStoreException at runtime.

**Ex:** class Parent {}

class Child extends Parent {}

class Unrelated {}

public class Test {

public static void main(String[] args) {

Child[] childArray = new Child[5];

Parent[] parentArray = childArray; // Allowed (Child[] → Parent[])

parentArray[0] = new Child(); // Allowed

parentArray[1] = new Parent(); // Runtime Error: ArrayStoreException

**(Note:** Storing a parent class object into a child class array causes ArrayStoreException, because the object type is not compatible with the array’s declared element type.)

}

} **(Arrays in Java are covariant, meaning Child[] can be assigned to Parent[].** However, runtime checks ensure that only Child objects can be stored in parentArray.**)**

**→** When you assign one array to another, the internal elements are not copied; instead, the reference variable is reassigned. This means both variables will point to the same array object in memory.

**Ex:** public class Test {

public static void main(String[] args) {

int[] arr1 = {10, 20, 30};

int[] arr2 = arr1; // Reference reassignment, NOT element copying

arr2[0] = 100; // Changing arr2 also affects arr1

System.out.println(arr1[0]); // Output: 100

System.out.println(arr2[0]); // Output: 100

}

} (To create a new independent copy, use clone(), Arrays.copyOf(), or manual copying)

**→** When assigning one array to another, the dimensions must match, or else a compilation error occurs. Only arrays of the same type and dimension can be assigned to each other.

**Ex:** int[] arr1 = {1, 2, 3};

int[][] arr2 = arr1; // CE: Incompatible types (1D ≠ 2D)

**(Not Allowed: Assigning arrays of different dimensions (int[] → int[][]))**

**→** Based on values, variables in Java are categorized into two types:

Primitive Variables → Store actual values (e.g., int, char, double).

Reference Variables → Store memory addresses (references) of objects. (They point to objects in the heap memory)

**Ex:** int a = 10; // Primitive variable (stores actual value)

String str = "Hello"; // Reference variable (stores address of String object)

**→** Based on their declaration position, variables in Java are categorized into three types: **1.** Instance Variables **2.** Static Variables **3.** Local Variables

**Instance Variables :**

Declared inside a class but outside any method, constructor, or block. Belong to an object and are created when an object is instantiated. **Stored in the heap memory.** Each object gets a separate copy of instance variables.

**Ex:** class Test {

int x = 10; // Instance variable

}

**Static Variables :**

Declared with the static keyword inside a class but outside any method, constructor, or block. Belong to the class rather than any object. **Stored in the method area (class area).** Shared among all objects of the class.

**Ex:** class Test {

static int y = 20; // Static variable

}

**Local Variables :**

Declared inside a method, constructor, or block. Exist only during the method execution. **Stored in the stack memory.** Must be explicitly initialized before use.

**Ex:** class Test {

void display() {

int z = 30; // Local variable

System.out.println(z);

}

}

**→** You cannot directly access instance variables from a static context because static areas belong to the class itself rather than to any specific instance. However, you can access instance variables using an object reference or you can access instance variables directly from instance areas.

**→** For instance variables, JVM provides default values, so it is not mandatory to initialize them before use. Instance variables are called object-level variables or attributes.

**Detailed Flow of JVM Execution :**

**JVM Startup:** The Java Virtual Machine (JVM) starts and initializes necessary components.

**Main Thread Creation:** JVM creates the main thread for execution.

**Class Loading:** The JVM locates the Test.class file using the class loader.

**Bytecode Verification:** The JVM verifies bytecode for security.

**Memory Allocation:**  JVM loads the class into the method area, initializes static members, and creates an object for

execution.

**Method Execution:** The main() method is invoked. If it creates other threads, they execute parallelly.

**Class Unloading:** Once execution completes, the class is removed from memory.

**JVM Shutdown:** When the main() method and all other threads complete, the JVM shuts down.

**→** A static variable can be accessed using either a class name or an object reference, but it is recommended to use the class name for better readability. Inside the same class, the class name is optional, and you can directly access the static variable.

**→** Static variables can be accessed directly from both static and instance areas of a class.

**Ex:** class Test {

static int x = 100; // Static Variable

static void staticMethod() {

System.out.println(x); // Direct access in static area

}

void instanceMethod() {

System.out.println(x); // Direct access in instance area

}

public static void main(String[] args) {

staticMethod();

Test obj = new Test();

obj.instanceMethod();

}

}

**→** For static variables, JVM provides default values, so it is not mandatory to initialize them before use. Static variables are called class-level variables or fields.

**→** Local variables are declared inside a method, constructor, or block. They must be initialized before use because the JVM does not provide default values. They are accessible only within the block where they are declared. Stored in stack memory and destroyed after method execution.

**→ It is not recommended to initialize local variables inside logical** blocks (e.g., if, switch) because these blocks may not execute at runtime, leading to "variable might not have been initialized" compilation errors. To avoid errors, always initialize local variables at the time of declaration, even with a default value.

**Ex:** class Test {

public static void main(String[] args) {

int x; // Not initialized

if (false) {

x = 10; // This block won't execute, x remains uninitialized

}

System.out.println(x); // CE: Variable 'x' might not have been initialized

}

}

**→** Java prevents access to uninitialized local variables by giving a compile-time error, instead of allowing garbage values.

**→ The only allowed access modifier for local variables is final.** Using any other access modifier (public, private, protected, or static) will cause a compilation error.

**Ex:** class Test {

public static void main(String[] args) {

final int x = 10; // Allowed

System.out.println(x);

}

}

**→** If no access modifier is specified for a instance/static variable, it has the default access modifier, known as **"package-**

**private."** This means it is accessible only within its own package. However, this concept of default access modifiers does not

apply to local variables declared within methods, as these variables are inherently only accessible within the method they

are defined in and do not have visibility outside of that scope.

**→** Instance and static variables can be accessed by multiple threads simultaneously, making them not thread-safe unless explicitly synchronized. Local variables are thread-safe because each thread gets its own separate copy in the stack memory, preventing interference between threads.

**→** In Java, variables can be categorized based on their storage type and data type. The possible combinations are:

Instance + Primitive Instance + Reference

Static + Primitive Static + Reference

Local + Primitive Local + Reference

**→** Once an array is created, its elements are automatically initialized with default values based on the declared type, regardless of whether the array is local, instance, or static.

**→** Instance and static arrays get default values (null reference initially). Local arrays do not get default values and must be explicitly initialized before use. Trying to access an uninitialized local array gives a compile-time error (CE). Trying to access an uninitialized instance/static array element gives a NullPointerException (NPE).

**→** A var-arg method (variable-length argument method) allows passing a variable number of arguments of the same type to a method. (Introduced in Java 5)

**Syntax:** returnType methodName(datatype... varName) {

// method body

}

**Ex:** class VarArgExample {

// Method with var-arg parameter

static void displayNumbers(int... numbers) {

for (int num : numbers) {

System.out.print(num + " ");

}

System.out.println();

}

public static void main(String[] args) {

displayNumbers(); // No arguments

displayNumbers(10, 20, 30); // Multiple arguments

}

}

**→** Advantages of Var-Arg Methods : Reduces method overloading: Instead of creating multiple overloaded methods for different numbers of parameters, a single var-arg method can handle all cases. Flexible argument passing: Allows passing zero or more arguments, making it more convenient. Improves code readability & maintainability.

**Rules for Var-Arg Methods:**

void test(int... a, int... b) {} // CE: Only one var-arg parameter allowed

void test(int... a, String s) {} // CE: Var-arg must be last

void test(String s, int... a) {} // Allowed

void test(int... a) {

System.out.println(a.length); // Internally, a var-arg parameter is treated as an array (int... a → int[] a)

}

**→** You cannot declare both a var-arg method (int...) and an array method (int[]) with the same parameter type inside the

same class. This results in a "duplicate method" compilation error(if both methods have the exact same signature).

**→** **Var-arg methods have the least priority** in method resolution. If there is an exact match or a widening/conversion match,

those methods will be chosen first. The var-arg method will be called only if no other method matches.

**Ex:** class Test {

static void m1(int a) {

System.out.println("Single int method");

}

static void m1(int... nums) {

System.out.println("Var-arg method");

}

public static void main(String[] args) {

m1(10); // Calls "Single int method" (Exact match)

m1(10, 20); // Calls "Var-arg method" (No exact match)

}

}

**→** You can replace an array method with a var-arg method because var-arg internally behaves like an array. However, the reverse is not always possible because the var-arg method can handle additional cases that an array method cannot. (The main method can be defined using either public static void main(String[] args) or public static void main(String... args))

**Ex:** class Test {

static void m1(String... args) { // Replacing String[] with String...

System.out.println("Var-arg method executed");

}

public static void main(String... args) {

m1(); // Works (Empty case)

m1("Hello"); // Works (Single argument)

m1("A", "B", "C"); // Works (Multiple arguments)

m1(new String[]{"X", "Y", "Z"}); // Works (Passing an array)

}

} **(Var-arg (String... args)** can fully replace an array method (String[] args). But an array method (String[] args) cannot fully replace a var-arg method (String... args))

**→** If you write void m1(int[]... x), the method will accept any number of one-dimensional integer arrays, and x will behave as a two-dimensional array (int[][]) internally.

**→** The compiler does not check for the presence or correctness of the main method. Instead, the JVM is responsible for verifying whether the main method is present and correctly declared.

The compiler only checks for syntax errors. The JVM looks for the main method at runtime to start execution. If the main method is missing or incorrectly declared, the JVM throws an error.

**Ex:** public class Test {

static void hello() {

System.out.println("Hello World");

}

} ( Compiles Successfully. JVM Error at runtime )

**Error: Main method not found in class Test**, please define the main method as: public static void main(String[] args)

**→** The main method in Java has a specific prototype that must be followed for the JVM to recognize and execute it:

public static void main(String[] args)

**Breakdown of the main Method Terminologies:**

(But changing access modifiers, return type, or making it non-static will result in a runtime error)

**public (Access Modifier):** The main method must be public so that the JVM (which is outside the class) can call it.

**static (Method Modifier):** The main method is static because it must run without creating an object of the class.

**void (Return Type):** The main method does not return any value to the JVM.

**main (Method Name):** The method name must be exactly main because the JVM looks for this specific name to start

execution.

**(String[] args) (Parameter):** The parameter is an array of strings **(String[] args) that holds command-line arguments.**

**→** The order of modifiers in the main method is not important. You can write: static public void main(String[] args) // Valid or public static void main(String[] args) // Valid

**→** In the main method, you can replace args with any valid Java identifier. **For example:** public static void main(String[] data)

**→** You can use final, synchronized, and strictfp modifiers in the main method. These modifiers do not affect the execution of the main method but are allowed by Java.

**Ex:** public final synchronized strictfp static void main(String[] args) {

System.out.println("Main method with all allowed modifiers.");

}

**→ Method overloading is possible for the main method in Java**, but the JVM always calls public static void main(String[] args). Other overloaded main methods must be called explicitly.

**Ex:** public class MainOverloading {

// JVM calls this main method

public static void main(String[] args) {

System.out.println("Main with String[] args");

}

public static void main(int a) {

System.out.println("Main with int: " + a);

}

public static void main(String s) {

System.out.println("Main with String: " + s);

}

}

**→ Inheritance is applicable** to the main method. If a child class does not have a main method, the parent class's main method will execute when running the child class.

**Ex:** class Parent {

public static void main(String[] args) {

System.out.println("Parent class main method");

}

}

class Child extends Parent {

// No main method in the child class

}

Running Child: $ java Child

Parent class main method // Output

**→** For the main method, method overloading and inheritance are applicable, but overriding is not applicable. Instead, it follows **method hiding because main is a static method.**

**Example of Method Hiding:** class Parent {

public static void main(String[] args) {

System.out.println("Parent main method");

}

}

class Child extends Parent {

public static void main(String[] args) { System.out.println("Child main method");

}

}

Running Child: $ java Child

Child main method // Output

**→** **From Java 7 onwards**, the main method is mandatory for program execution or else the JVM will throw an error, even if a static block is present, it will not execute if the main method is missing. So it's impossible to print anything to the console without the main method.

**Example (Before Java 7 - Worked in Java 6):**

class Test {

static {

System.out.println("Static block executed");

System.exit(0); // Exiting without main method

}

}

Output (Java 6 or earlier): Static block executed

**Example (Java 7 and later):**

class Test {

static {

System.out.println("Static block executed");

}

}

Error (Java 7 and later): Error: Main method not found in class Test, please define the main method as:

public static void main(String[] args)

**Java Execution Flow (Java 7 and Later):**

**Check for main method:** The JVM first checks if the main method (public static void main(String[] args)) is present. If it is missing, the program will not execute, and an error will be thrown.

**Identification of Static Members:** The JVM identifies all static variables and static blocks in the class.

**Assignment of Static Variables:** All static variables are assigned to their default values (e.g., int → 0, boolean → false, String → null).

**Execution of Static Blocks:** Static blocks are executed in the order they appear in the class. If multiple static blocks are present, they execute sequentially.

**Execution of main Method:** Once all static members are initialized, the JVM calls the main method to start program execution.

**→** Command-line arguments are the arguments passed to the main method when executing a Java program from the terminal or command prompt. They are passed as strings and can be accessed using the String[] args parameter of the main

method. JVM creates the String[] args array and then calls the main method, passing the command-line arguments as

elements of this array.

**→** The main purpose of command-line arguments is to customize the behavior of the main method by passing input values at runtime.

**→** In the main method, command-line arguments are passed as String values. To use them as other data types, we must

explicitly convert them using wrapper classes and their parse methods.

**Ex:** public class CommandLineExample {

public static void main(String[] args) {

if (args.length < 2) {

System.out.println("Please provide two numeric arguments.");

return;

}

// Converting String arguments to int and double

int num1 = Integer.parseInt(args[0]);

double num2 = Double.parseDouble(args[1]);

System.out.println("Sum: " + (num1 + num2));

}

}

**→** Space is the separator between command-line arguments. If an argument contains spaces, you must enclose it in double quotes.

**Example :** java CommandLineSpace Hello "Java Programming" World

(The argument "Java Programming" is treated as a single argument because it is enclosed in double quotes)

**→** When writing Java code, it's highly recommended to follow coding standards. The name of a component must reflect its purpose or functionality.

**Advantages:** Improves Readability – Makes it easier for developers to understand the code.

Enhances Maintainability – Helps in modifying or debugging the code efficiently.

**Ex:** class Employee { // Class names should be in PascalCase

private String employeeName; // Variable names should be in camelCase

private int employeeId;

public void setEmployeeName(String name) { // Method names should be in camelCase

this.employeeName = name;

}

public String getEmployeeName() {

return employeeName;

}

} (Following these standards ensures that the code is clear, consistent, and easier to work with)

**Java naming conventions :**

**Class Names :** Nouns in PascalCase. Example: StudentRecord, BankAccount

**Interface Names :** Typically, interfaces are often nouns or noun phrases describing a capability.

Example: Comparable, Readable, Serializable

**Method Names :** Verb-noun combinations in camelCase.

Example: calculateTotalAmount(), validateUserCredentials()

**Variable Names :** Nouns in camelCase. Example: studentName, totalBalance

**Constant Names :** Uppercase with underscore separators for multiple words.

Example: MAX\_ARRAY\_SIZE, DEFAULT\_TIMEOUT

**→** Constant variables in Java are typically declared using public static final modifiers to create class-level constants that are accessible, shared across instances, and immutable after initialization.

**Syntax:** public static final data\_type CONSTANT\_NAME = value; Ex: public static final double PI = 3.14159;

**→** **A JavaBean** is a standard Java class that follows specific conventions: Must have a no-argument constructor. Properties should be private. Properties accessed via public getter and setter methods. Implements Serializable interface. (optional but recommended)

**Ex:** public class Student implements Serializable {

private String name;

public Student() {} // No-argument constructor

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

**→** For boolean properties in JavaBeans, the recommended getter method prefix is **'is'** instead of 'get'. This provides more natural and readable method names for boolean attributes. Using 'is' prefix (e.g., isEmpty(), isValid()) enhances code readability for boolean properties.

**→** In Java listener registration methods: Methods to add a listener are prefixed with add. Methods to remove a listener are prefixed with remove.

**Example pattern:** public void addActionListener(ActionListener listener)

public void removeActionListener(ActionListener listener)

(This convention ensures clear, consistent method naming for event listener management)

**Operators & Assignments**

**→** Increment (++) and decrement (--) operators can only be used with **variables.** Using them with constant values will result in a compilation error (CE).

**Ex:** int x = 5;

x++; // Valid

5++; // Compilation Error (CE)

**(Constants like 5++ are not allowed because they are not variables whose values can be modified)**

**→** Increment (++) and decrement (--) operators are used to increase or decrease a variable's value by 1.

Pre-Increment (++x) : Increments the value first, then uses it.

Post-Increment (x++) : Uses the value first, then increments it.

Pre-Decrement (--x) : Decrements the value first, then uses it.

Post-Decrement (x--) : Uses the value first, then decrements it.

**→** You cannot apply increment (++) or decrement (--) operators to final variables in Java because final variables are constant and cannot be re-assigned after their initial assignment.

**→** You can apply increment and decrement operators to primitive types that represent numeric values, specifically byte,

short, int, long, float, double, and char.

**→** In Java, when performing an operation on two different data types, the result follows this rule:

**Result Data Type = max(int, Type1, Type2)**  For example:

byte + byte → int short + short → int char + char → int

byte + short → int int + float → float long + float → float

float + double → double

**→** If you want the result in a specific type, you must explicitly typecast it.

**Ex:** byte a = 10, b = 20;

byte c = (byte) (a + b); // Explicit typecasting required

System.out.println(c); // Output: 30

**→** Increment (++) and decrement (--) **operators internally perform typecasting** to ensure the final result matches the operand type.

**Ex:** byte b = 10;

b++; // No need for explicit typecasting

System.out.println(b); // Output: 11

(However, b++ is internally converted to b = (byte) (b + 1), ensuring type safety)

**→** Java provides a set of arithmetic operators for performing mathematical calculations on numeric values. ( + (Addition), - (Subtraction), \* (Multiplication), / (Division), % (Modulo) )

**→** In integral arithmetic, division by zero (10/0) causes a runtime exception (ArithmeticException), as integers do not support infinity. However, in floating-point arithmetic, Java defines two constants: **Double.POSITIVE\_INFINITY** and **Double.NEGATIVE\_INFINITY**

So, 10 / 0.0 → Infinity -10 / 0.0 → -Infinity 10 / 0 → ArithmeticException: / by zero

**→** In integral arithmetic, 0/0 causes a runtime exception (ArithmeticException: / by zero) because Java does not support an "undefined" value for integers. However, **in floating-point arithmetic, Java defines the constant NaN (Not a Number) for undefined results.**

So, 0 / 0.0 → NaN -0 / 0.0 → NaN 0 / 0 → ArithmeticException: / by zero

**→** Any comparison with Float.NaN (<, >, <=, >=, ==) **always returns false**. **Ex:** 10 <= Float.NaN → false

Float.NaN == Float.NaN → false

**However, inequality (!=) always returns true.** **Ex:** 10 != Float.NaN → true (This is because NaN is not equal to anything, including itself)

**→** ArithmeticException is a runtime exception, not a compile-time error. It only occurs in integral arithmetic (Ex: int, long). It does not occur in floating-point arithmetic because floating-point operations handle division by zero using Infinity and NaN. The only operators that cause ArithmeticException are: / (division) and % (modulus)

**→ The only overloaded operator in Java is the + operator.** As an arithmetic operator: When used with numeric types, it performs addition. (10 + 20 → 30)

**As a concatenation operator:** When used with a String, it performs string concatenation. ("Hello" + " World" → "Hello World") ( If one operand is a String, **Java converts the other operand to a String** and performs concatenation instead of addition.)

**→** You can apply relational operators to all primitive numeric types (byte, short, int, long, float, double, char). While you can technically use relational operators with **char (because it's represented numerically),** You cannot directly apply relational operators to boolean types. You would use logical operators (&&, ||, !) to combine or manipulate boolean values.

**→** Using relational operators (<, >, <=, or >=) with objects will result in a compile-time error. **You can use relational operators (== and !=) with object types, but they compare references (memory addresses),** not the object's content.

**→** Nesting relational operators directly **(Ex: x < y < z) is not allowed in Java.**

**→** Equality operators (== and !=) can be applied to all primitive types, including boolean. For booleans, == checks if two booleans are the same, and != checks if they are different.

**→** When using the equality **(==) operator** to compare two objects: There **must be a relationship between them** (either

same type or parent-child). If no relation exists, it results in a CE.

**Ex:** String s = new String("Hello");

Object o = new Object();

System.out.println(s == o); // Valid (String is a subclass of Object)

String s1 = "Hello";

Integer i = 10;

System.out.println(s1 == i); **// CE (String and Integer have no relation)**

**Difference between == operator and equals() method in Java :**

**Feature == Operator equals() Method**

Comparison Type Compares references (memory addresses) Compares content

(data inside objects)

Applicable for Both primitives and objects Only for objects

(not for primitives)

Default Behavior For objects, checks if they Default (Object class) **checks**

refer to the same memory location  **reference like ==,** but can be

overridden (Ex: in String, Integer)

**Ex:** String s1 = new String("Hello");

String s2 = new String("Hello");

System.out.println(s1 == s2); // false (Different objects)

System.out.println(s1.equals(s2)); // true (Same contents)

**(Use == when checking memory reference and equals() when checking actual content)**

**→** When you compare any object reference with null using ==, it returns false if the reference is not null and true if it is null. Additionally, null == null always returns true.

**Ex:** String s1 = null;

String s2 = "Hello";

System.out.println(s1 == null); // true

System.out.println(s1 == s2); // false

**→** **The instanceof** operator is used to check whether an object is an instance of a specific class or implements a specific interface. It returns true if the object is an instance of the given type, otherwise, it returns false. ( Syntax: object instanceof ClassName )

**Ex:** class Parent {}

class Child extends Parent {}

public class InstanceofExample {

public static void main(String[] args) {

Parent p = new Parent();

Child c = new Child();

Parent p2 = new Child();

System.out.println(p instanceof Parent); // true

System.out.println(c instanceof Parent); // true (Child is a Parent)

System.out.println(p instanceof Child); // false

System.out.println(p2 instanceof Child); // true

}

} (Prevents ClassCastException in downcasting. Returns false if the reference is null, i.e., **null instanceof AnyClass is always false)**

**→ When using the instanceof operator, there must be a relationship** between the two types (either parent-child or the same class). Otherwise, it results in a compile-time error (CE).

**Ex:** class A {}

class B {}

public class Test {

public static void main(String[] args) {

A a = new A();

System.out.println(a instanceof B); // CE: incompatible types

}

} **( instanceof works safely within an inheritance hierarchy )**

**Bitwise Operators in Java:**

**Bitwise AND (&):** Performs a bitwise AND operation between two numbers. The result is 1 if both corresponding bits are 1,

otherwise 0.

**Example:** int a = 5; // 0101

int b = 3; // 0011

System.out.println(a & b); // Output: 1 (0001)

**Bitwise OR (|):** Performs a bitwise OR operation between two numbers. The result is 1 if at least one corresponding bit is 1,

otherwise 0.

**Example:** int a = 5; // 0101

int b = 3; // 0011

System.out.println(a | b); // Output: 7 (0111)

**Bitwise XOR (^):** Performs a bitwise XOR (exclusive OR) operation. The result is 1 if the corresponding bits are different, otherwise 0.

**Example:** int a = 5; // 0101

int b = 3; // 0011

System.out.println(a ^ b); // Output: 6 (0110)

( These operators are mainly used for **low-level programming, encryption, and performance optimization** techniques )

**→** These operators can be applied to both integral types (byte, short, int, long) and boolean types.

**Logical Operations (Boolean Types):** When applied to booleans, they behave as logical operators.

**Ex:** boolean x = true;

boolean y = false;

System.out.println(x & y); // Output: false (true AND false)

System.out.println(x | y); // Output: true (true OR false)

System.out.println(x ^ y); // Output: true (true XOR false)

**→** **The bitwise NOT (~)** operator is only applicable to integral types (byte, short, int, long). If applied to boolean values, it results in a compile-time error (CE).

**Example with Integers:** int x = 5; // Binary: 0000 0101

System.out.println(~x); // Output: -6 (Inverts all bits)

**( Explanation: ~5 results in -6 because of two’s complement representation )**

**Example with Boolean:** boolean flag = true;

System.out.println(~flag); // CE: bad operand type boolean for unary operator '~'

**→** **The logical NOT (!)** operator is only applicable to boolean types. If applied to integral types, it results in a compile-time error (CE).

**Ex:** boolean isJavaFun = true;

System.out.println(!isJavaFun); // Output: false

int x = 5;

System.out.println(!x); // CE: bad operand type int for unary operator '!'

**Difference Between Bitwise (&, |) and Logical (&&, ||) Operators:**

**Feature Bitwise &, | Logical &&, ||**

Usage Works on both integral and boolean types Works only on boolean types

Short-Circuit No short-circuiting, both operands Short-circuiting happens: second

are always evaluated operand is evaluated only if needed

Example int result = 5 & 3; → Performs bitwise AND boolean result = (x > 5) && (y++ > 10); boolean res = true & false; → y++ executes only if x > 5 (true)

→ Evaluates both expressions boolean res = true || false;

→ Second operand not evaluated

**→** **Implicit Typecasting** (also known as automatic type promotion) occurs when the Java compiler automatically converts a

smaller data type into a larger data type without explicit instruction from the programmer. The Java compiler is responsible for implicit typecasting.

Possible Implicit Typecasting: byte → short → int → long → float → double

char → int (since char is internally stored as an integer)

**Ex:** int a = 10;

double b = a; // int to double (implicit)

char c = 'A'; // ASCII value of 'A' is 65

int d = c; // char to int (implicit)

System.out.println(d); // Output: 65

(Key Points: Implicit typecasting avoids data loss by converting to a larger type. It does not work for narrowing conversions (e.g., double to int). Ensures type safety without additional syntax)

**→** **Explicit Typecasting** (also known as manual type conversion) occurs when a larger data type is explicitly converted into a smaller data type using typecasting syntax. This process is done by the programmer and may lead to data loss if the target type cannot hold the original value completely.

Possible Explicit Typecasting: double → float → long → int → short → byte

int → char (Narrowing conversion), float → int (Fractional part is truncated)

**Ex:** double d = 10.99;

int i = (int) d; // double to int (explicit)

System.out.println(i); // Output: 10 (fractional part lost)

int num = 130;

byte b = (byte) num; // int to byte (explicit)

System.out.println(b); // Output: -126 (due to overflow)

float f = 97.6f;

char ch = (char) f; // float to char (explicit)

System.out.println(ch); // Output: 'a' (ASCII 97)

(May cause data loss or overflow (e.g., int to byte). Fractional parts are lost when casting floating-point numbers to integers. Used in interfacing incompatible types (e.g., Object to String))

**Types of Assignment Operations in Java :**

**Simple Assignment** (=) : Assigns a value to a variable. Example: int a = 10;

**Chained Assignment** (= used multiple times) : Assigns the same value to multiple variables in a single statement.

**Example:** int x, y, z;

x = y = z = 20; // All three variables get the value 20

Compound Assignment (+=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=): Combines arithmetic or bitwise operation with assignment.

**Example:** int num = 10;

num += 5; // num = num + 5 → 15

num \*= 2; // num = num \* 2 → 30

num >>= 1; // num = num >> 1 → 15 (right shift)

( **Performs implicit typecasting when needed** (e.g., byte += int doesn't require explicit casting) )

**→** **>> (Signed Right Shift):** This operator shifts the bits of a number to the right. The leftmost bits (the sign bits) are filled with the original sign bit. So, if the number was positive, it fills with 0s. If it was negative, it fills with 1s. This preserves the sign of the number.

**→** **>>> (Unsigned Right Shift or Zero-Fill Right Shift):** This operator also shifts the bits to the right, but it always fills the leftmost bits with 0s, regardless of the original sign bit. This is often called a "zero-fill" right shift. It's important to note that this operator can result in a positive value even if the original value was negative, because the sign bit is not preserved.

**Ex:** int x = -8; // Binary representation (32 bits): 11111111111111111111111111111000

int y = x >> 2; // y will be -2 (Binary: 11111111111111111111111111111110) - Sign bit (1) is propagated

int z = x >>> 2; // z will be 1073741822 (Binary: 00111111111111111111111111111110) - Zeros are filled

System.out.println("x >> 2: " + y); // Output: -2

System.out.println("x >>> 2: " + z); // Output: 1073741822

**→** You can combine chained assignments and compound assignments in Java. For example:

int x, y, z;

x = y = z = 10; // Chained assignment: x, y, and z all get the value 10

x = y += 2; // Combined: y becomes 12 (y = y + 2), and then x gets the new value of y (12)

System.out.println("x: " + x); // Output: 12

System.out.println("y: " + y); // Output: 12

System.out.println("z: " + z); // Output: 10

**→** The only ternary operator in Java is the Conditional Operator (? :). It acts as a shorthand for if-else. You can nest ternary operators inside another.

**Ex:** int a = 10, b = 20;

int min = (a < b) ? a : b; // min = 10

int x = 5, y = 10, z = 15;

int min = (x < y) ? (x < z ? x : z) : (y < z ? y : z); // min = 5

**→** The primary purpose of the **new** operator is to create a new object in Java. Ex: String s = new String("Hello");

**→ The constructor does not create a new object; it initializes the object after creation.**

**Ex:** class Demo {

Demo() {

System.out.println("Constructor Called");

}

}

Demo d = new Demo(); // Object is created, then constructor runs

**→** Java does not have a delete keyword because the JVM’s Garbage Collector (GC) automatically manages memory. Objects that are no longer referenced are eligible for garbage collection.

**→** **[] (square brackets)** is an operator in Java, used for: Array Declaration & Initialization, Array Indexing (Accessing Elements)

Ex: int[] numbers = {10, 20, 30};

System.out.println(numbers[1]); // Output: 20

**( It is an operator because it allows direct access to array elements using an index )**

**→** **Here is the operator precedence in Java (from highest to lowest):**

Postfix → expr++, expr--

Unary → ++expr, --expr, +, -, ~, !

Type Cast → (type)

Multiplicative → \*, /, %

Additive → +, -

Shift → <<, >>, >>>

Relational → <, <=, >, >=, instanceof

Equality → ==, !=

Bitwise AND → &

Bitwise XOR → ^

Bitwise OR → |

Logical AND → &&

Logical OR → ||

Ternary (Conditional) → ?:

Assignment → =, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=

( Note: Operators with higher precedence are evaluated first, and operators with the same precedence follow left-to-right associativity (except for assignment and ternary operators, which are right-to-left )

**→** In Java, operator precedence determines the order in which operators are applied, but operand precedence does not exist. Before any operator executes, all operands are evaluated from left to right, regardless of operator precedence.

Ex: System.out.println(10 + 20 \* getNumber()); ( If getNumber() is a method, it will be evaluated first before applying \* or + )

**→** the new operator and newInstance() method are both used to create objects in Java, but they differ in how and when they are used. Key Differences:

**Feature new Operator newInstance() Method**

Usage When the class name is known When the class name is determined

at compile time at runtime

Syntax ClassName obj = new ClassName(); Class obj = Class.forName("ClassName"); obj.newInstance(); (Java 8 and earlier)

Checked Exception No exception handling required Requires handling ClassNotFoundException,

IllegalAccessException and InstantiationException

Constructor Invocation Directly calls the constructor Calls the no-arg constructor of the class

Deprecation Not deprecated Class.newInstance() is deprecated in Java 9+, use **getDeclaredConstructor().newInstance()** instead

**Ex:** public class Test {

@SuppressWarnings("deprecation")

public static void main(String[] args) throws Exception {

Object obj = Class.forName(args[0]).newInstance();

System.out.println("Object created for class : "+obj.getClass().getName());

}

}

**→** If you use the **new operator** and the class was compiled but is not available at runtime, the JVM throws NoClassDefFoundError. This typically happens when the .class file exists during compilation but is missing or corrupted at runtime. (Runtime Error) Ex: Exception in thread "main" **java.lang.NoClassDefFoundError: MissingClass**

**→** If you use **newInstance() method** and the class is not found in the classpath, a ClassNotFoundException is thrown. This happens because the JVM tries to dynamically load the class but cannot locate it. (Checked Exception) Example: **ClassNotFoundException: MissingClass**

**→** Both instanceof operator and isInstance() method are used to check whether an object is an instance of a specific class or its subclass, but there are key differences between them:

**Feature instanceof Operator isInstance() Method**

Definition A Java operator used to check if an object A method in Class that performs a

is an instance of a class or subclass similar check dynamically

Usage object instanceof ClassName Class.forName(args[0]).isInstance(object)

Compile-time Check Yes, checked at compile-time No, checked at runtime

Null Handling null instanceof AnyClass returns false Class.isInstance(null) returns false

When to Use When the type is known at compile-time When the type is determined at runtime

Checked Exception No exception handling required Requires handling ClassNotFoundException

**Flow-Control**

**→** Flow control determines the order in which statements execute. The main types of flow control statements in Java are:

Selection Statements – if, if-else, switch

Iterative Statements – for, while, do-while

Transfer Statements – break, continue, return, throw, try-catch-finally

**→** The expression inside the parentheses of an **if statement must evaluate to a boolean value (either true or false).** If you try to use a non-boolean expression, you will get a compile-time error.

**→** **The else block is indeed optional in an if statement.** Curly braces {} are optional only if the if block (or the else block, if present) contains only a single statement. If curly braces are omitted, only the immediately following statement is considered part of the if block (or else block). **A declaration statement is not allowed** directly after an if (or else) without curly braces. This will result in a compile-time error. Declarations must be within a block (using curly braces).

**→** **A semicolon by itself (;)** is a valid Java statement, often called the empty statement or null statement. It doesn't do anything, but it's syntactically correct.

**→** Java does not have the **"dangling else"** problem that can occur in some other languages (like C/C++). In Java, an else block is always associated with the nearest preceding unmatched if block in the same scope. The compiler enforces this rule, so there's no ambiguity.

**→** If multiple options are available, it's not recommended to use if-else statements as they reduce readability. In such cases, using a switch statement is preferable. A switch statement is used to execute one block of code among multiple options based on the value of an expression.

**→** **The switch statement supports the following data types as its argument:**

1. Primitive data types: byte, short, char, int
2. Wrapper classes:Byte, Short, Character, Integer
3. Enumerations (enum)
4. String (String)

( long, float, double, and their wrapper classes are not allowed. The expression inside a switch can be **any valid expression.** switch supports both traditional (case:) and enhanced (case ->) syntax (introduced in Java 12) )

**Traditional Switch Statement Example:**

public class SwitchExample {

public static void main(String[] args) {

int number = 2;

switch (number) {

case 1: System.out.println("One");

break;

case 2: System.out.println("Two");

break;

case 3: System.out.println("Three");

break;

default: System.out.println("Invalid number");

}

}

}

(**Explanation:** Each case block ends with a break statement to **prevent fall-through.** The default case executes when no case matches the value.)

**→ The outer curly braces {} are mandatory in a switch statement.** if-else and loops (for, while, do-while) allow single statements without {}, but switch always requires {}.

**→** case and default are optional – A switch statement can be completely empty without any case or default. Independent statements are NOT allowed inside switch – Any statement must be inside a case or default block; otherwise, it results in a compilation error.

**Example 1: Empty switch (Valid)**

public class Test {

public static void main(String[] args) {

int num = 3;

switch (num) { // Valid but does nothing

}

}

} // Compiles successfully but has no effect.

**Example 2: Independent Statements Inside switch (Invalid)**

public class Test {

public static void main(String[] args) {

int num = 2;

switch (num) {

System.out.println("Hello"); // Compilation Error: Statement not allowed

case 1: System.out.println("One");

break;

case 2: System.out.println("Two");

break;

}

}

} **// Compilation Error: Independent statements are not allowed inside switch.**

**→** Both the switch argument and case label can be expressions. **The case label must be a constant expression, meaning it must evaluate to a compile-time constant.**

**Example 1: Valid Case – switch Argument as an Expression**

public class Test {

public static void main(String[] args) {

int a = 2, b = 3;

switch (a + b) { // switch argument as an expression

case 5: System.out.println("Sum is 5");

break;

default: System.out.println("Not 5");

}

}

} // Output: Sum is 5

**Example 2: Valid Case – case Label as a Constant Expression**

public class Test {

public static void main(String[] args) {

final int x = 10;

switch (x) {

case 5 + 5: System.out.println("Matched 10"); // Constant expression (evaluates to 10 at compile-time)

break;

default: System.out.println("No match");

}

}

} // Output: Matched 10

**Example 3: Invalid Case – case Label as a Non-Constant Expression**

public class Test {

public static void main(String[] args) {

int x = 10;

int y = 10;

switch (x) {

case y: System.out.println("Matched"); // Compilation Error: 'case' label must be a constant expression

break;

}

}

} **// Error: case label must be a compile-time constant, but y is a variable.**

**Key Takeaways :** The switch argument can be an expression. The case label must be a constant expression, meaning: It must be final if using a variable. It must be computable at compile-time (like 5 + 5). It cannot be a regular variable (int y = 10; case y: is invalid)

**→** Every case value **must be within the valid range** of the switch argument type; otherwise, a compilation error occurs.

**Invalid Case – case Value Out of Range (Compilation Error):**

public class Test {

public static void main(String[] args) {

byte num = 10;

switch (num) {

case 128: System.out.println("Out of range"); // Compilation Error: 128 is out of byte range (-128 to 127)

break;

}

}

} // If a case value exceeds the type's range, it results in a compilation error.

**→** Duplicate case labels within the same switch block are not allowed in Java and will result in a compile-time error. **Each case label must have a unique value.**

**→ Fall-through occurs when a case does not have a break**, allowing execution to continue into the next case. Main Advantage of Fall-Through: Reduces code duplication when multiple cases share the same logic.

**Example: Grouping Cases Using Fall-Throughs**

public class Test {

public static void main(String[] args) {

int num = 1;

switch (num) {

case 1:

case 2:

case 3: System.out.println("Number is between 1 and 3");

break;

default: System.out.println("Other number");

}

}

} // Advantage: All cases 1, 2, and 3 execute the same code without repetition.

**→** **You can have at most one default label in a switch statement.** It is executed only when no other case matches. While you can technically place the default label anywhere within the switch, it's strongly recommended to place it at the end for clarity and readability.

**→** The while loop is the appropriate choice when you don't know the number of iterations in advance. The loop continues to execute as long as the condition remains true.

**Syntax:** while (condition) {

// Code to be executed repeatedly as long as the condition is true

}

( condition is a boolean expression that is evaluated before each iteration. If the condition is true, the loop body is executed. If the condition is false, the loop terminates. The condition inside the parentheses of a while loop must be a boolean expression. Any other data type will result in a compile-time error.)

**→** If you omit the curly braces {} in a while loop, only the immediately following statement is considered part of the loop's body. **That single statement cannot be a declaration.** A declaration must be inside a block (defined by curly braces). Attempting to declare a variable directly after a while without braces will result in a compile-time error.

**→** Using true or false directly in a while condition leads to an unreachable statement error if there's any code after the loop. Comparing two final variables in a condition (final int a = 10, b = 20; while (a < b) {}) causes a compilation error, because the condition is constant and known at compile time. Using normal (non-final) variables works fine, as the condition is evaluated at runtime. **Ex:**

public class Test {

public static void main(String[] args) {

while (true) {

System.out.println("Infinite Loop");

}

System.out.println("Unreachable Code"); // Compilation Error

}

} // Unreachable code error occurs if while (true) {} is used without a break.

**→ The value of a final variable (if it is a compile-time constant) is replaced at compile time.** Any operation performed on two final variables (or any compile-time constants) is evaluated during compilation.

**→** A do-while loop is an exit-controlled loop, meaning the loop executes at least once, regardless of the condition. The

condition is checked after executing the loop body. **Syntax:**

do {

// Loop body (executes at least once)

} while (condition);

(Condition Checked After Execution: Unlike while, where the condition is checked first. **Semicolon (;) Required: The while (condition); must end with a semicolon**.)

**→** In a do-while loop, the curly braces {} are optional only if the loop body contains a single statement. And that single statement cannot be a declaration. Declarations require a block (curly braces). Trying to put a declaration directly between do and while without curly braces will lead to a compile-time error.

**→ Identifying unreachable statements is the responsibility of the compiler, not the JVM (Java Virtual Machine).** The compiler performs static analysis of the code and detects situations where a statement can never be reached during program execution. The JVM executes the compiled bytecode; it doesn't do the kind of analysis needed to find unreachable code.

**→** A for loop is an entry-controlled loop used for iterating a fixed number of times. It consists of three parts: initialization, condition, and increment/decrement. **Syntax:**

for (initialization; condition; update) {

// Loop body

}

Initialization → Executes once before the loop starts.

Condition → Checked before each iteration; if false, the loop stops.

Update → Executes after each iteration (increment/decrement).

**Ex:** public class Test {

public static void main(String[] args) {

for (int i = 1; i <= 5; i++) {

System.out.println("Iteration: " + i);

}

}

}

**Key Points:** Best for fixed iteration loops when the number of iterations is known. **All three parts are optional** (can be omitted for infinite loops). Can have multiple initializations or updates, separated by commas. Loop body can be empty if logic is in the update part.

**Example: Infinite for Loop**

for (;;) { // No condition = runs forever

System.out.println("Infinite Loop");

} // Runs indefinitely unless interrupted (break).

**→** The rules for curly braces in a for loop are the same as for while and do-while loops. If the loop body contains only a single statement, the curly braces {} are optional. And that single statement (if braces are omitted) cannot be a declaration.

**→** In the initialization section of a for loop, you can declare multiple variables, but they must be of the same data type. If you try to declare variables of different data types, it will result in a compilation error (CE).

public class Test {

public static void main(String[] args) {

for (int i = 1, j = 5; i <= 5; i++, j--) { **// Valid**

System.out.println("i: " + i + ", j: " + j);

}

}

}

public class Test {

public static void main(String[] args) {

for (int i = 1, double d = 2.5; i <= 5; i++) { // CE: Different data types

System.out.println(i + " " + d);

}

}

} // To use different data types, declare them outside the loop

**→** Any valid Java statement can be placed in the initialization section, including method calls and System.out.println(). If a statement in the initialization section does not declare a variable, the loop remains valid. The statement executes once before the loop starts.

public class Test {

public static void main(String[] args) {

int i = 1;

for (System.out.println("Hello"); i <= 10; i++) { // Valid

System.out.println(i);

}

}

}

**→** Any valid expression of boolean type can be written in the condition section of a for loop. The condition is optional – if omitted, **it defaults to true,** making the loop infinite unless terminated with a break statement.

public class Test {

public static void main(String[] args) {

for (;;) { // No condition → Defaults to ‘true’ (Infinite loop)

System.out.println("Infinite Loop");

break; // To prevent actual infinite execution

}

}

} (Condition must be a boolean expression (true, false, or a comparison like i < 10). Non-boolean expressions (e.g., i = 5) in the condition will cause a compilation error)

**→** The increment/decrement section of a for loop allows any valid Java statement, including method calls like System.out.println(). It executes after each iteration of the loop body. **Example:**

public class Test {

public static void main(String[] args) {

int i = 1;

for (System.out.println("Hello"); i <= 5; System.out.println("Hi")) { // Valid

i++;

}

}

}

**→** **The for-each loop** (also called enhanced for loop) simplifies iteration over collections and arrays. It eliminates the need for explicit indexing, reducing errors like ArrayIndexOutOfBoundsException. **Introduced in Java 5. Syntax:**

for (DataType variable : collection/array) {

// Code to be executed for each element

}

**Ex1:** import java.util.\*;

public class Test {

public static void main(String[] args) {

List<String> names = Arrays.asList("Alice", "Bob", "Charlie");

for (String name : names) {

System.out.println(name);

}

}

}

**Ex2:** int[][][] arr = { { { 1, 2, 3 }, { 4, 5, 6 } }, { { 7, 8, 9 }, { 1, 0, 8 } } }; // Iterating Over a 3D-Array

for (int[][] x1 : arr) {

for (int[] x2 : x1) {

for (int i : x2) {

System.out.println(i);

}

}

}

( Simplifies iteration by avoiding index-based loops. Works with arrays and collections (List, Set, Map). **Cannot modify collection size (adding/removing elements will throw ConcurrentModificationException).** Cannot access index (use for loop if index-based operations are needed).

**→** The enhanced for loop (or "for-each" loop) is specifically designed for iterating over arrays and collections (anything that implements the Iterable interface). However, it's not a general-purpose loop like the traditional for loop. You cannot use it in situations where you need more control over the iteration process (e.g., manipulating indices, iterating in reverse, or skipping elements). It's purpose-built for simple, element-by-element traversal of arrays and collections.

**→** The Iterable interface was introduced in Java 1.5 (Java 5). It does indeed have only one method: iterator(). All collection classes (ArrayList, HashSet, LinkedList, etc.) implement the Iterable interface directly or indirectly. This allows them to be used in a for-each loop and work with iterators. **Arrays (int[], String[], etc.) DO NOT implement Iterable.** However, you can still use the enhanced for loop with arrays. The Java compiler handles arrays specially in this case, even though they don't implement the interface. It's a bit of compiler magic that makes the enhanced for loop **work with both Iterable objects (Collections) and arrays.**

**Difference Between Iterator and Iterable in Java:**

**Feature Iterable<T> Iterator<T>**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Definition Represents a collection that can Represents a cursor to iterate over be iterated over. elements oneby one.

Belongs To Collection (List, Set, etc.) that A separate object used to supports iteration. traverse a collection.

Located In java.lang package. java.util package.

Key Methods iterator() (returns an Iterator). hasNext(), next(), remove().

Usage Implemented by collection classes Used to iterate elements from (ArrayList, HashSet, etc.) an Iterable object.

State Represents the whole collection. Maintains iteration state (cursor position).

**Example of Iterable and Iterator:**

import java.util.\*;

public class Test {

public static void main(String[] args) {

List<String> names = new ArrayList<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

// Using Iterable (For-Each Loop)

for (String name : names) { // Implicitly uses Iterator

System.out.println(name);

}

// Using Iterator Explicitly

Iterator<String> iterator = names.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

}

}

**Key Takeaways:**

**Iterable<T> →** Represents a collection that can be iterated (supports for-each).

**Iterator<T> →** Provides methods to iterate (hasNext(), next(), remove()).

All collection classes (List, Set, Queue) implement Iterable. **To manually iterate, use Iterator.**

**→** The break statement in Java is used to terminate the execution of a loop (for, while, do-while) or a switch statement prematurely. When a break statement is encountered inside a loop or a switch, the loop or switch is immediately exited, and control transfers to the statement immediately following the loop or switch.

**Where can we use it?**

**Loops**: break is commonly used within loops to exit the loop based on a certain condition.

**Switch Statements**: break is essential in switch statements to prevent "fall-through." Without a break at the end of each case (or default), execution would continue into the next case.

**Labeled Break** (Less Common): The break statement in Java, when used with a label, allows you to exit any labeled block of code, not just loops or switch statements. A labeled block is simply a block of code (enclosed in curly braces {}) that has a label associated with it.

**Ex:** outerBlock: { // Labeling a block of code

innerBlock: {

System.out.println("Inside inner block");

if (someCondition) {

break outerBlock; // Exit the outer block

}

System.out.println("Still inside inner block"); // This might not execute

}

System.out.println("Inside outer block"); // This might not execute

}

System.out.println("After outer block"); // This will always execute

In this example, if someCondition is true, the break outerBlock; statement will immediately transfer control to the statement after the outerBlock (the System.out.println("After outer block"); line).

**→** If you use a break statement outside of a switch statement, a loop (for, while, do-while), or a labeled block, you will get a compile-time error.

**→** The continue statement is used to skip the current iteration of a loop and jump to the next iteration.

**Where can we use continue?**

Inside Loops (for, while, do-while).

With Labels (to skip iterations of an outer loop).

Not Allowed in if-else or switch statements.

**Example :** continue with a Label (Skipping Outer Loop Iteration)

public class Test {

public static void main(String[] args) {

outer:

for (int i = 1; i <= 3; i++) {

for (int j = 1; j <= 3; j++) {

if (i == 2 && j == 2) {

continue outer; // Skips the rest of iteration of outer loop when i=2, j=2

}

System.out.println(i + " " + j);

}

}

}

} // Explanation: When i == 2 && j == 2, the outer loop jumps to its next iteration.

**→** You can use a break statement with a label to exit an outer loop from within an inner loop. This is less frequently used but can be helpful in nested loop scenarios.

**Ex:** outerLoop:

for (int i = 0; i < 5; i++) {

for (int j = 0; j < 5; j++) {

if (i == 2 && j == 3) {

break outerLoop; // Exit the outer loop

}

System.out.println("i: " + i + ", j: " + j);

}

}

**→** In a do-while loop, when the continue statement is encountered, control does not go directly to the top of the loop. Instead, it jumps to the condition check of the while statement. If the condition is true, the loop continues; otherwise, it terminates.

**Example: continue in a do-while Loop**

public class Test {

public static void main(String[] args) {

int i = 1;

do {

if (i == 3) {

i++; // Increment before continue to avoid infinite loop

continue;

}

System.out.print(i+” ”);

i++;

} while (i <= 5);

}

} // Output: 1 2 4 5 (i = 3 is skipped, but the loop does not restart from the top immediately)

**Declarations and Access Modifiers**

**====================================================================================**

**→** A Java source file can contain any number of classes (inner classes, helper classes, etc.), but it can have **at most one public class**. If there is a public class, the name of the source file (the .java file) must match the name of that public class exactly (including case). If there is no public class, the name can be anything. If these rules are violated, the compiler will issue an error.

**→** If a Java source file contains multiple classes, separate .class files will be generated for each class during compilation. The name of each .class file will match the name of the corresponding class. You compile a Java source file (e.g., javac Filename.java), which generates .class files for all the classes in the file. **You can only run a .class file (e.g., java ClassName), not the source file directly.**

**→** When executing a .class file, the JVM looks for the main method in the corresponding class. Only the class containing the

main method can be executed directly. If the main method is missing, the JVM will throw a runtime error (Error: Main method not found in class). If the .class file is missing or corrupted, the JVM will throw a runtime error **(ClassNotFoundException or NoClassDefFoundError).**

**→** During compilation, the Java compiler checks for dependencies between classes. **If one class depends on another, both will be compiled, and their .class files will be generated.**

**→** It is not recommended to define multiple classes in a single Java source file. Instead, it is highly recommended to define only one class per source file.

**Benefits of Single Class per File**:

**Readability:** Keeping one class per file makes the code easier to read and understand.

**Maintainability:** It simplifies maintenance, as changes to a class are isolated to its own file.

**Organization:**  It improves project organization, making it easier to locate and manage classes.

If a class is small and tightly coupled with another class (e.g., a helper or utility class), it can be defined in the same file as an non-public class.

**Best Practices:**

Always name the source file after the public class it contains. Use meaningful and consistent naming conventions for both files and classes. Group related classes into packages to further improve organization and maintainability.

**→** Using fully qualified names (e.g., java.util.ArrayList) increases code length and reduces readability.

**Example:** java.util.ArrayList<String> list = new java.util.ArrayList<>(); **Solution:**

**Import Statements:** To avoid using fully qualified names, you can use import statements at the beginning of your Java file.

**Example:** import java.util.ArrayList;

public class Main {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<>();

}

}

**Benefits of Import Statements:**

Improved Readability: Code becomes cleaner and easier to read.

Reduced Redundancy: Eliminates the need to repeatedly type long package names.

Better Maintainability: Makes the code easier to update and manage.

**Types of Import Statements:**

**→** Single-Type Import: Imports a specific class from a package. **Ex: import java.util.ArrayList**;

**→** On-Demand Import: Imports all classes from a package using \*. **Ex: import java.util.\*;**

**(Note:** While convenient, using \* can sometimes lead to ambiguity if multiple packages contain classes with the same name**)**

**Default Imports:** Some packages are automatically imported in every Java program, such as: **java.lang** (e.g., String, System, Math). No explicit import is needed for these classes.

**Static Imports:** You can use static imports to import static members (fields and methods) of a class.

**Example:** import static java.lang.Math.PI;

import static java.lang.Math.pow;

public class Main {

public static void main(String[] args) {

double radius = 5.0;

double area = PI \* pow(radius, 2);

System.out.println("Area of circle: " + area);

}

}

Avoiding Naming Conflicts: If two classes with the same name are imported from different packages, you must use the fully qualified name for at least one of them to resolve the conflict.

**Example:** import java.util.Date;

import java.sql.Date;

public class Main {

public static void main(String[] args) {

java.util.Date utilDate = new java.util.Date();

java.sql.Date sqlDate = new java.sql.Date(0);

}

}

**→ Explicit Imports:** These are imports where you specify the exact class you want to import (e.g., import java.util.ArrayList;). This is generally considered best practice because it makes the code easier to understand and maintain. It's clear which classes are being used.

**Implicit Imports (Wildcard Imports):** These use an asterisk \* to import all classes within a package (e.g., import java.util.\*;). They are sometimes convenient, but they can make the code less clear, especially in larger projects. It's not immediately obvious which specific classes are being used.

**→ import java.util.ArrayList.\*;** is invalid because using .\* after a specific class name wouldn't make sense; wildcards are for packages, not individual classes. **import java.util;** is also invalid. While java.util is a valid package, import statements require either a specific class name (e.g., java.util.ArrayList) or the wildcard \* to indicate all classes within the package (e.g., java.util.\*). **Importing just the package name by itself is not allowed.**

**→** If you use the fully qualified name of a class (including the package name) throughout your code, then you do not need an import statement for that class. The fully qualified name provides the compiler with all the necessary information to locate the class. Conversely, if you use an import statement (either explicit or implicit), you can then use the simple class name (without the package) in your code, and you don't need to use the fully qualified name. They are mutually exclusive ways of referring to a class.

**→** java.util.Date and java.sql.Date: Both classes represent dates, but they serve different purposes (one for general dates and times, the other specifically for database interactions). java.util.List and java.awt.List: java.util.List is the interface for ordered collections (lists), while java.awt.List is a GUI component. If you were to use an implicit import (import java.util.\*; and import java.sql.\*; or import java.awt.\*;) and then just use Date or List in your code, the compiler wouldn't know which class you intended to use, leading to a **compile-time error (ambiguous class name).**

**When resolving a class name, the Java compiler follows this order of precedence:**

**Explicit Import**: The compiler first checks if there's an explicit import statement for the specific class name (e.g., import java.util.ArrayList;). If found, this has the highest priority.

**Current Directory** (Unnamed Package/Default Package): If no explicit import is found, the compiler then looks for a .class file (or .java file, which it would then compile) with the given name in the current directory. This is effectively the unnamed package or default package.

**Implicit Import** (Wildcard Import): If the class is still not found, the compiler then searches the packages specified by implicit import statements (e.g., import java.util.\*;)

**→** Importing a package (e.g., import java.util.\*;) makes the classes and interfaces directly within that package available. It does not automatically import classes or interfaces from subpackages. If you want to use classes from a subpackage (e.g., java.util.concurrent), you must import that subpackage explicitly (e.g., import java.util.concurrent.\*; or import the specific class you need import java.util.concurrent.ConcurrentHashMap;). Package imports are not transitive in Java.

**→ The java.lang package** is implicitly imported into every Java program. You never need to write an import java.lang.\*;

statement. Classes and interfaces within java.lang (like String, Object, System, Thread, etc.) are always available. Similarly, **classes in the current working directory** (the unnamed package or default package) are also implicitly available. The compiler searches the current directory when resolving class names. So, if you have a class in the same directory as your code, you don't need an import statement to use it.

**→** Import statements are indeed primarily a compile-time concept. They help the compiler locate and resolve the classes and interfaces used in your code. The number of import statements has a very minor impact on compilation time but import statements have no direct impact on execution time. The JVM doesn't care about import statements. Once the code is compiled, the import information is no longer relevant.

**→** C's #include is a preprocessor directive that performs a textual inclusion of the header file's contents at compile time. It's a static inclusion. The compiler effectively copies and pastes the contents of the header file into the C source file before compilation. Java's import statement, on the other hand, is not a textual inclusion. It's a declaration to the compiler that tells it where to look for classes and interfaces during compilation. The .class files are not loaded at compile time. They are loaded by the JVM at runtime, on demand, as they are needed. This is dynamic loading.

**→ Java 5 (JDK 1.5) New Features:** Generics, Enhanced for-loop (foreach loop), Autoboxing and Unboxing, Varargs (Variable Arguments), Static Imports, Annotations (Metadata), Enums, Concurrency Utilities (java.util.concurrent package), Formatter and Scanner classes, New java.util.Queue and Deque interfaces

**→ With static import,** you can directly access static members (methods or variables) without using the class name.

Example: With Static Import

import static java.lang.Math.sqrt; // Static import

public class Test {

public static void main(String[] args) {

double result = sqrt(25); // No need for Math.sqrt()

System.out.println(result);

}

}

// Key Benefit: No need to prefix static members with the class name. Improves readability and reduces redundancy.

**If you static import java.lang.System.out;** you can then use out.println() directly in your code, rather than System.out.println(). It's as if out becomes directly available as a variable in your current scope.

**What is System.out.println();**

System.out.println(); is a Java statement used to print text or data to the console (standard output). It is one of the most commonly used methods for debugging, displaying messages, or outputting results in Java programs. **Breaking Down System.out.println();**

**System:** System is a final class in the java.lang package. It provides access to system-level resources, such as standard input, output, and error streams.

**out:** out is a **static member** of the System class. It is an object of the **PrintStream class**, which represents the standard output stream (typically the console). By default, out is connected to the console, so anything sent to out is displayed on the screen.

**println:** println is a method of the PrintStream class. It prints the specified data to the console and adds a newline character (\n) at the end, moving the cursor to the next line. (There is also a similar method called print, which does not add a newline character)

**How Does It Work?** When you call System.out.println(); the following happens:

The System class provides access to the out object (standard output stream). The println method of the PrintStream class is invoked on the out object. The data passed to println is printed to the console, followed by a newline.

**Syntax: System.out.println(data);** (data: The content to be printed. It can be a string, number, variable, expression, or even nothing (to print a blank line))

**The order of precedence for resolving static members (methods or variables) in Java:**

Current Class: The compiler first looks for a static member with the given name in the current class (or interface). This has

the highest priority.

Explicit Static Import: If not found in the current class, the compiler checks for an explicit static import of that specific member (e.g., import static java.lang.System.out;)

Implicit Static Import (Wildcard Static Import): If still not found, the compiler then searches the classes specified by implicit static imports (e.g., import static java.lang.System.\*;)

**→** Ambiguity issues in normal import are rare because having classes or interfaces with the same name in different packages is uncommon. However, ambiguity is more frequent in static import since methods and variables with the same name exist in multiple classes. **Drawbacks of Static Import:**

Reduces Readability **→** It becomes unclear which class a static member belongs to.

Creates Confusion **→** If multiple static members have the same name, ambiguity arises.

Not Recommended Unless Necessary **→** It should be used only when it significantly improves code clarity.

(**Best Practice:** Prefer using ClassName.method() instead of static import to maintain code clarity and avoid confusion)

The fundamental distinction between regular imports and static imports:

**Regular Imports:** Used to import classes and interfaces from a package, allowing you to use them by their simple name (without the package prefix).

**Static Imports:** Used to import static members (methods or variables) of a specific class, allowing you to use them directly by their name (without the class name prefix).

**What is a Package in Java?**

A package in Java is a mechanism for organizing related classes, interfaces, and sub-packages into a single namespace. It helps in:

Organizing Code: Grouping related classes together for better structure.

**Avoiding Naming Conflicts:** Preventing class name collisions by providing a unique namespace.

Access Control: Providing access control through package-private (default) visibility.

Reusability: Making it easier to reuse code across projects.

Packages are hierarchical, meaning they can have sub-packages (e.g., java.util, java.util.concurrent).

**Syntax of a Package:** To declare a package, use the **package keyword** at the top of a Java file:

Ex: package com.example.mypackage; (The directory structure must match the package name. For example, the above package should be in the folder: com/example/mypackage/)

**Types of Packages :**

Built-in Packages: Predefined packages provided by Java (e.g., java.lang, java.util, java.io).

User-defined Packages: Packages created by developers to organize their own code.

**Here are some commonly used built-in packages in Java:**

1. **java.lang:** Provides fundamental classes and interfaces for the Java language. Automatically Imported. No need to explicitly import this package.

Examples of Classes: String: For string manipulation.

System: For system-level operations (e.g., System.out.println).

Math: For mathematical operations (e.g., Math.sqrt, Math.PI).

**2. java.util:** Contains utility classes and data structures.

Examples of Classes: ArrayList: A resizable array implementation.

HashMap: A key-value pair collection.

Scanner: For reading user input.

Date: For date and time operations.

**3. java.io:** Provides classes for input and output operations.

Examples of Classes: File: Represents a file or directory.

FileReader: Reads text from files.

FileWriter: Writes text to files.

BufferedReader: Reads text efficiently.

**4. java.sql:** Provides classes for database connectivity and operations.

Examples of Classes: Connection: Represents a connection to a database.

Statement: Executes SQL queries.

ResultSet: Holds the results of a database query.

**→** Java has well-established naming conventions for packages, designed to ensure uniqueness and avoid conflicts.

Here's a breakdown:

**Reverse Domain Name Notation:** The most common and recommended convention is to use a reversed domain name as the base package name. For example, if your company's domain is example.com, your packages would typically start with com.example. This helps to ensure global uniqueness because domain names are themselves unique.

Further Subpackages: Within the base package, you can create subpackages to organize your code further. These subpackage names should be descriptive and reflect the purpose of the code within them. For example, com.example.myapp.model, com.example.myapp.ui, com.example.myapp.util.

( Package names should be all lowercase. This is a strong convention. Underscores are generally discouraged in package names. Package names should be as descriptive as possible, giving a clear indication of the code they contain )

**→** When you specify a package in a Java program, you must compile it using the -d option to ensure that the .class file is placed in the correct package structure within the current working directory.

Compilation Command: **javac -d . FileName.java**

**-d . →** Places the compiled .class file inside the correct package folder in the current directory (.)

**Example: Java File: Test.java**

package com.example;

public class Test {

public static void main(String[] args) {

System.out.println("Hello, Package!");

}

}

Compilation: javac -d . Test.java

Generated Folder Structure: ./com/example/Test.class

(**Key Takeaway: -d .** ensures proper package structure. The .class file is placed inside the corresponding package directory. **Run the program using java com.example.Test** So, while you can organize your .class files using -d during compilation, you must use the fully qualified name during execution so that the JVM can locate the correct class)

**→** If the specified package structure does not exist, the command javac -d . FileName.java automatically creates the required directory structure and places the .class file inside it. You can replace **.** with any valid directory path. If the directory doesn't exist, javac will create it (and any necessary parent directories). **For example: javac -d bin Test.java** Compiles Test.java and places the Test.class file in a directory named bin (which will be created if it doesn't exist in the current directory)

**→** The javac -d command can create folders (directories) but not drive letters (e.g., C:, D:, E:). If you specify a non-existent

drive like Z: (which does not exist), you will get a Compilation Error (CE) because javac cannot create a new drive.

Invalid Usage (Drive Does Not Exist): javac -d Z: Test.java // If Z: is not a valid drive, you get a compilation error. (**Key Takeaways:** javac -d can create directories but not drives. The specified path must exist up to the drive level (e.g., C: must

exist))

**→** In a Java source file (.java file), you can have **at most one package** declaration. The package declaration, if present, must be the very first non-comment and non-whitespace line in the file. It specifies the package to which the class or interface defined in that file belongs.

**The order of statements in a Java source file is important and follows this structure:**

1. Package Statement (at most one, optional)
2. Import Statements (any number, optional)
3. Class/Interface/Enum Declarations (any number, optional)

**Explanation:** Package statement (if present) must always be the first statement. Import statements must come after the package and before class/interface declarations. **A Java source file can be empty** (i.e., no class/interface), and it will still compile without errors.

**Example of a Proper Java File:**

package com.example; // 1st: Package statement (optional)

import java.util.\*; // 2nd: Import statements (optional)

public class Test { // 3rd: Class declaration (optional)

public static void main(String[] args) {

System.out.println("Hello, Java!");

}

}

**Class-Level Modifiers in Java:**

Class-level modifiers (also known as access modifiers or non-access modifiers) are keywords used to define the access level and behavior of a class in Java. They determine how a class can be accessed and used by other classes and packages.

**Types of Class-Level Modifiers:**

1. **Access Modifiers:** These control the visibility and accessibility of a class. There are four access levels in Java.

Modifier Description:

**public:** The class is accessible from any other class in any package.

**default (no modifier):** The class is accessible only within the same package. This is also called package-private.

**protected:** Not applicable for top-level classes. It is used for class members (fields, methods).

**private:** Not applicable for top-level classes. It is used for class members.

**Example:**

public class MyClass { } // Accessible everywhere

class MyClass { } // Accessible only within the same package

**2. Non-Access Modifiers:** These define additional behavior or properties of a class.

Modifier Description:

**final:** The class cannot be subclassed (inherited).

**abstract:** The class cannot be instantiated and is meant to be subclassed.

**strictfp:** Ensures floating-point calculations are platform-independent.

**Examples:**

final class MyFinalClass { } // Cannot be inherited

abstract class MyAbstractClass { } // Cannot be instantiated

strictfp class MyStrictClass { } // Ensures consistent floating-point calculations

Rules for Using Class-Level Modifiers:

**→ Only one access modifier** can be used for a class (e.g., public, default). Multiple non-access modifiers can be used together (e.g., final and strictfp). A class cannot be both final and abstract because they have contradictory meanings.

(**Key Points:** A top-level class (not nested inside another class) can only have public or default access. A class cannot be **private or protected** at the top level. Use final to prevent inheritance and abstract to enforce subclassing.)

**→ The list of applicable modifiers for inner classes:** public, protected, private, (default), static, abstract, final, strictfp

**→ The final modifier** can be applied to classes, methods, and variables in Java.

**→ final class:** A final class cannot be subclassed (inherited from). This is often used to prevent unintended modification or extension of a class. **String** **is a well-known example of a final class.**

**→ final method:** A final method cannot be overridden in subclasses. This is used to prevent subclasses from changing the behavior of a method.

**→ final variable:** A final variable can be assigned a value only once. After it's initialized, its value cannot be changed. If the final variable is a reference to an object, the reference itself cannot be changed (it will always point to the same object), but the contents of the object can still be modified if the object is mutable. If the final variable is a primitive type, its value is immutable. final variables are often used for constants.

**→** If a method is available in the parent class, it is inherited by the child class by default. If the child class is not satisfied with the parent class's implementation, it can override the method. This is called method overriding. However, if the parent class method is declared final, it cannot be overridden in the child class. **Ex:**

class Parent {

final void display() { // Final method (cannot be overridden)

System.out.println("Parent class method");

}

}

class Child extends Parent {

// Compilation Error: Cannot override final method from Parent

/\*

@Override

void display() {

System.out.println("Child class method");

}

\*/

}

( Method overriding allows the child class to provide a new implementation of the parent class method. A final method in the parent class cannot be overridden in the child class. Overriding is only possible for **instance methods** (non-static). **Access modifiers must be the same or more permissive** (e.g., protected method in parent can be protected or public in child).)

**→** If a class is declared as final, it cannot be extended, meaning inheritance is not possible. **Ex: Integer Class** (The Integer class in Java is declared as final, meaning it cannot be extended)

// Compilation Error: Cannot inherit from final 'Integer'

class MyInteger extends Integer {

} // Since Integer is final, Java prevents subclassing to maintain immutability and avoid unexpected behavior.

( **Key Takeaways:** A final class cannot be extended. String, Integer, Double, and many other wrapper classes in Java are final for security and immutability. Marking a class as final ensures that no one can modify its behavior via inheritance )

**→** Every method in a final class is **implicitly final.** Since a final class cannot be subclassed, there's no possibility of overriding any of its methods. Variables within a final class are not automatically final. You must explicitly declare a variable as final if you want it to be immutable.

**→ Advantages of final:** Enhances security by preventing modification. Ensures immutability for variables, methods, and classes. Provides compile-time optimization. Helps in creating constant values.

**→ Limitations and Considerations:** Prevents inheritance for final classes. Restricts method overriding for final methods. Can limit flexibility in object-oriented design.

**→ Recommendation:** Only when there's a specific security or design requirement. Avoid overusing to maintain OOP principles like inheritance and polymorphism. If there's no compelling reason to use final, it's better to keep the code open for extension and customization.

**Abstract Modifier in Java:**

The abstract modifier is used to define classes and methods that are incomplete and must be implemented or extended by subclasses. It is a way to achieve abstraction in Java, allowing you to define a blueprint for other classes without providing a complete implementation.

Key Features of the abstract Modifier:

**For Classes:** An abstract class cannot be instantiated (you cannot create objects of an abstract class). It is meant to be subclassed (inherited) by other classes. It can contain both abstract methods (methods without a body) and concrete methods (methods with a body).

**For Methods:** An abstract method does not have a body (no implementation). It must be overridden (implemented) by the subclass. **If a class contains even one abstract method, the class itself must be declared abstract.** (This is because the presence of an abstract method signifies that the class is incomplete. It defines a contract (the abstract method signature) but doesn't provide a full implementation. Therefore, it wouldn't make sense to create an object of such a class directly because the object would be missing the implementation of that method.)

**Example of an Abstract Class:**

abstract class Animal {

// Concrete method

void sleep() {

System.out.println("This animal sleeps.");

}

// Abstract method (no body)

abstract void sound();

}

Here, Animal is an abstract class with: A concrete method sleep(). An abstract method sound().

**When you extend an abstract class, you must:** Override all abstract methods in the subclass. If you don’t override all abstract methods, the subclass must also be declared abstract. **Example:**

class Dog extends Animal {

// Override the abstract method

@Override

void sound() {

System.out.println("The dog barks.");

}

}

**Key Points About Abstract Classes:** You cannot create an object of an abstract class. **Ex:** Animal animal = new Animal(); // Error: Animal is abstract; cannot be instantiated

**Can Have Constructors:** Abstract classes can have constructors, which are called when a subclass is instantiated. Ex:

abstract class Animal {

Animal() {

System.out.println("Animal constructor called.");

}

}

**Can Have Fields and Concrete Methods:** Abstract classes can have fields, concrete methods, and even static methods.

**Can Extend Another Class:** An abstract class can extend another class (abstract or concrete).

**Key Points About Abstract Methods:**

**No Body:** Abstract methods do not have a body (implementation). Ex: abstract void sound(); // No curly braces {}

**Must Be Overridden:** If a subclass is not abstract, it must override all abstract methods of the parent class.

**Cannot Be private or final:** Abstract methods cannot be private (they must be accessible to subclasses) or final (they must be overridable).

**Abstract Class vs. Interface:**

**Feature Abstract Class Interface**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Instantiation Cannot be instantiated. Cannot be instantiated.

Methods Can have both abstract and concrete methods. Can have abstract, default, and static methods (Java 8+).

Fields Can have fields (variables). Can only have public static final fields (constants).

Multiple Inheritance Does not support multiple inheritance. Supports multiple inheritance.

Constructor Can have constructors. Cannot have constructors.

**→** An abstract method only defines the method signature (what to do) but never provides an implementation (how to do it). Since abstract means "must be overridden," and some modifiers conflict with this, **the following modifier combinations are illegal: final, native, synchronized, static, private, strictfp.**

**→** The main purpose of an abstract class, even one without abstract methods, is to serve as a base class for other classes. It might define a common interface or provide some shared implementation that subclasses can inherit. Even if it has no abstract methods, it can still have concrete methods and variables that are relevant to its subclasses. Here are some relevant examples of abstract classes without any abstract methods: **HttpServlet** (javax.servlet.http.HttpServlet), **GenericServlet** (javax.servlet.GenericServlet), **InputStream** (java.io.InputStream), **OutputStream** (java.io.OutputStream)

**→ final + abstract combination is illegal** both at the class level and method level because final prevents inheritance/overriding, while abstract requires inheritance/overriding.

**→** **An abstract class can contain both abstract and final methods.** However, a final class, by definition, cannot have any abstract methods, as it cannot be subclassed to provide an implementation for those abstract methods.

**→** It's highly recommended to use the abstract modifier judiciously, as it facilitates key Object-Oriented Programming (OOP)

concepts like inheritance, polymorphism, and abstraction. By declaring a class or method as abstract, you're explicitly

indicating that it's intended to be inherited and customized by subclasses, which helps promote code reusability, modularity,

and extensibility.

**strictfp Modifier in Java:**

**→ The strictfp** (strict floating-point) modifier is used to ensure consistent floating-point calculations across all platforms. It guarantees that the results of floating-point operations are the same on every platform, regardless of the underlying hardware or operating system.

Why is strictfp Needed?

**→** Floating-point calculations can vary slightly between different platforms due to differences in hardware and how floating-point numbers are handled. The strictfp modifier enforces **IEEE 754 floating-point standards,** ensuring platform independent results.

The strictfp modifier can be applied to:

Classes: Ensures all floating-point calculations in the class follow IEEE 754 standards.

Methods: Ensures floating-point calculations in the method follow IEEE 754 standards.

Interfaces: Ensures all floating-point calculations in the interface follow IEEE 754 standards.

**Example for a Class:**

strictfp class MyClass {

// All floating-point calculations in this class are strictfp

}

**Example for a Method:**

class MyClass {

strictfp void myMethod() {

// Floating-point calculations in this method are strictfp

}

}

**Example for an Interface:**

strictfp interface MyInterface {

// All floating-point calculations in this interface are strictfp

}

**Key Points About strictfp:**

**Platform Independence:** Ensures consistent floating-point results across all platforms.

**Performance Impact:** Using strictfp may slightly reduce performance because it restricts optimizations that rely on

platform-specific floating-point behavior.

**Scope:** If applied to a class, all methods and nested classes in that class will follow strictfp behavior. If applied to a method, only that method will follow strictfp behavior.

**→** Not Commonly Used: Most applications do not require strictfp unless they need strict consistency in floating-point calculations (e.g., scientific or financial applications).

**Example of strictfp** in a Real-World Scenario:

Suppose you are writing a financial application where precision in floating-point calculations is critical. You can use strictfp to ensure consistent results across all platforms.

strictfp class FinancialCalculator {

double calculateInterest(double principal, double rate, double time) {

return principal \* rate \* time;

}

}

public class Main {

public static void main(String[] args) {

FinancialCalculator calculator = new FinancialCalculator();

double interest = calculator.calculateInterest(1000.0, 0.05, 2.0);

System.out.println("Interest: " + interest); // Output: Interest: 100.0

}

} // Use strictfp, In applications where precision is critical, such as scientific, engineering, or financial software.

**→** The combination of abstract and strictfp modifiers for a method is illegal and will result in a compile-time error because it's a logical inconsistency; you can't have an abstract method (no implementation) that is also strictfp (related to implementation).

**→ A class can be both abstract and strictfp.** In this case:

**abstract:** The class cannot be instantiated directly.

**strictfp:** All floating-point calculations within the class and any nested classes are performed according to the strict IEEE 754

rules. (So, the combination is valid at the class level)

**Method and Variable Level Modifiers in Java:**

Method and variable level modifiers (also known as access modifiers and non-access modifiers) are keywords used to define the access level and behavior of methods and variables in Java. They control how methods and variables can be accessed and used within a class, package, or program.

Types of Modifiers:

**1. Access Modifiers:** These control the visibility and accessibility of methods and variables. There are four access levels in Java, Modifier Description:

**public:** Accessible from any other class in any package.

**protected:** Accessible within the same package and by subclasses (even in other packages).

**default (no modifier)**: Accessible only within the same package. This is also called package-private.

**private:** Accessible only within the same class.

1. **Non-Access Modifiers:** These define additional behavior or properties of methods and variables.

Modifier Description:

**static:** Belongs to the class rather than an instance. Can be accessed without creating an object.

**final:** For variables: Value cannot be changed (constant). For methods: Cannot be overridden.

**volatile:** Ensures the variable is always read from and written to main memory (used in multithreading).

**transient:** Excludes the variable from serialization.

**native:** Indicates the method is implemented in platform-specific code (e.g., C/C++).

**strictfp:** Ensures consistent floating-point calculations across platforms.

**synchronized:** Ensures only one thread can execute the method at a time.

**abstract** For methods: No implementation (must be overridden in a subclass).

**Example 1: Access Modifiers**

public class MyClass {

public int publicVar = 1; // Accessible everywhere

protected int protectedVar = 2; // Accessible in the same package and subclasses

int defaultVar = 3; // Accessible only in the same package

private int privateVar = 4; // Accessible only in the same class

public void publicMethod() { }

protected void protectedMethod() { }

void defaultMethod() { }

private void privateMethod() { }

}

**Example 2: Non-Access Modifiers**

public class MyClass {

static int staticVar = 10; // Belongs to the class

final int finalVar = 20; // Cannot be changed

volatile int volatileVar = 30; // Always read from main memory

transient int transientVar = 40; // Excluded from serialization

static void staticMethod() { } // Belongs to the class

final void finalMethod() { } // Cannot be overridden

synchronized void syncMethod() { } // Thread-safe

strictfp void strictMethod() { } // Consistent floating-point calculations

}

Key Points to Remember:

**Access Modifiers:** Control the visibility of methods and variables. private is the most restrictive, while public is the least restrictive.

**Non-Access Modifiers:** Define additional behavior or properties. For example, static makes a method or variable belong to the class, and final makes it unchangeable.

**Combining Modifiers:** You can combine access and non-access modifiers (e.g., public static, private final). However, some combinations are invalid (e.g., abstract and final cannot be used together).

**Best Practices:** Use private for encapsulation and data hiding. Use final for constants or to prevent method overriding. Use static for utility methods or class-level variables.

**Class Visibility:** The compiler first checks the visibility of the class itself. If the class is not accessible from a particular location (e.g., if it's package-private and accessed from another package, or if it's private and accessed from outside the enclosing class), then no members of that class can be accessed from that location, regardless of the member's access modifier.

**Member Visibility:** Only after the class visibility is established does the compiler then consider the access modifier of the member itself (public, protected, private, or default). The member's visibility is then checked within the context of the class's accessibility.

**→** An abstract method must be implemented by a concrete subclass. However, a private method is not accessible to subclasses. Therefore, a subclass could never provide the required implementation for a private abstract method, that’s why **the combination of private and abstract modifiers** for a method is illegal in Java and will result in a compile-time error.

**→ protected members can be accessed:** Within the same class (where they are declared). Within the same package (accessible to other classes in the same package). In subclasses (even if they are in different packages) using inheritance. Outside the package, protected members can only be accessed through inheritance and not by direct reference. Example:

class A (in package1) has a protected member.

class B (in package2) extends A.

You can access the protected member using a reference of type B:

B b = new B();

b.protectedMethod(); // Correct

A a = new B(); // 'a' is a reference of type A

a.protectedMethod(); // Compile-time error!

Even though a refers to a B object, the reference type is A, and the compiler will not allow access. This is because protected access from outside the package is specifically designed to work through the inheritance relationship (subclass reference), not just through object instantiation.

**→ private is strongly recommended as the default access modifier for instance variables (fields).** This enforces encapsulation, which is a fundamental principle of OOP. **public is often** the appropriate access modifier for methods that are intended to be part of the class's public interface—methods that other classes will call.

**→** The order of access restriction (or conversely, the order of increasing accessibility) for Java's access modifiers: **private > (default/package-private) > protected > public**

**→** If an instance variable is declared as final, it must be explicitly initialized, because the JVM does not assign default values to final variables.

**→** A final instance variable (a field declared with the final keyword) must be initialized exactly once, and this initialization must occur before the constructor completes. The compiler enforces this rule. Here are the ways you can initialize a final instance variable:

**Directly in the declaration:** You can initialize the final variable at the point where it's declared:

private final int myValue = 10;

**In the constructor:** You can initialize the final variable within the constructor of the class:

private final int myValue;

public MyClass(int value) {

myValue = value;

}

// If you have multiple constructors, you must initialize the final variable in every constructor.

**In an initializer block (less common):** You can also use an initializer block to initialize a final variable:

private final int myValue;

{

myValue = 20;

}

(The key point is that the final variable must be initialized by the time the object's construction is finished. Attempting to initialize a final instance variable anywhere else (like inside a regular method) will indeed result in a compile-time error. The compiler will detect that you are trying to assign a value to a final variable after its initialization has already taken place (or outside the allowed contexts) and will issue an error. The rule is strictly enforced.)

**→** If a static variable is declared as final, it must be explicitly initialized, because the JVM does not assign default values to final variables. A static final variable (a constant) must be initialized before the class loading process is complete. This is because static final variables belong to the class itself, not to instances of the class, and they need to be available as soon as the class is loaded.

**→** Here are the ways you can initialize static final variables:

**Directly in the declaration:** This is the most common and preferred way. You initialize the constant at the point of declaration: private static final int MAX\_VALUE = 100;

**In a static initializer block:** You can also use a static initializer block to initialize a static final variable. This is useful if the initialization logic is more complex:

private static final int FACTORIAL\_10;

static {

int result = 1;

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

result \*= i;

}

FACTORIAL\_10 = result;

}

(These are the only valid locations for initializing static final variables. Attempting to initialize them anywhere else (like in a constructor or a regular method) will lead to a compile-time error)

**→** Even if a local variable is declared final, you are not required to actually use it in your code. The compiler doesn't care if a final local variable is declared but never used. A final local variable must be initialized before it's used. The compiler will enforce this.

**→** Local variables (variables declared inside a method or block) cannot have access modifiers. The keywords public, protected, and private are not allowed for local variables. Local variables have method-level or block-level scope. Their

visibility is inherently limited to the block of code where they're declared, so access modifiers are not applicable. Trying to use an access modifier on a local variable will result in a compile-time error.

**→** Formal parameters of a method act as local variables of that method. **They can be declared as final**, and if declared as final, they cannot be reassigned inside the method. Reassigning a final parameter inside the method leads to a compilation error. final parameters prevent modification within the method. **Useful for ensuring method arguments remain unchanged.**

**→** You can declare static methods and variables, but you can't declare a top-level class as static. However, **an inner class can be declared as static,** these are called static nested classes. When a variable is declared static, only one copy of that variable is created, regardless of how many objects of the class are instantiated. This single copy is shared by all instances of the class.

**→** We can't access instance members (variables or methods) directly from a static context because static belongs to the class, not an instance. However, we can access **static members** from both static and instance areas directly or by using the class name. But **instance members** can only be accessed using an object reference.

**→** **The name of a static or instance variable can be the same as the name of a local variable.** However, inside a method or constructor, the local variable takes precedence over the instance/static variable with the same name. **In this case:**

To access the instance variable, use **this.variableName.**

To access the static variable, use **ClassName.variableName.**

**→ Method overloading applies to static methods, including the main method.** However, JVM will always call the main(String[] args) method as the entry point of the program. Inheritance applies to static methods, including the main method. However, static methods do not participate in method overriding; they are hidden (method hiding). **If a child class does not define a main method, the parent class’s main method is invoked and executed** when running the child class.

**→** If a method uses at least one instance variable, it is related to an object and should be declared as an instance method. If a method does not use any instance variables, it is not tied to a specific object, so it is recommended to declare it as static, regardless of whether it uses static variables or not. This improves clarity and avoids unnecessary object creation.

**→** A static method must have an implementation, whereas an abstract method cannot have an implementation. This contradiction makes **the combination of static and abstract illegal in Java.**

**synchronized Modifier in Java:**

The synchronized modifier is used to **control access to a method or block** of code by multiple threads. It ensures that only one thread can execute the synchronized method or block at a time, preventing race conditions and ensuring thread safety.

Why is synchronized Needed?

**In a multithreaded environment,** multiple threads may try to access and modify shared resources simultaneously, leading to inconsistent or incorrect results. The synchronized modifier prevents this by allowing only one thread to execute the synchronized code at a time.

Where Can synchronized Be Used?

**Instance Methods:** Synchronizes the entire method on the current instance of the class (this). Only one thread can execute the method for a given instance of the class.

**Static Methods:** Synchronizes the entire method on the class object (not the instance). Only one thread can execute the method for the entire class.

**Code Blocks:** Synchronizes a specific block of code on a given object. Provides finer-grained control over synchronization.

**Synchronized Static Method:** public static synchronized void myStaticMethod() {

// Thread-safe code

}

**Synchronized Block:** public void myMethod() {

synchronized (this) { // Synchronizes on the current instance

// Thread-safe code

}

}

**How synchronized Works:**

**-----------------------------------**

When a thread enters a synchronized method or block, it acquires a lock on the object or class. Other threads must wait until the lock is released before they can execute the synchronized code. Once the thread exits the synchronized method or block, the lock is released, and another thread can acquire it.

With synchronized: Only one thread can execute the increment method at a time, ensuring thread safety. **Ex:**

class Counter {

int count = 0;

synchronized void increment() {

count++; // Thread-safe

}

}

Synchronized blocks allow you to synchronize only the critical section of the code, improving performance. **Example:**

class Counter {

int count = 0;

void increment() {

synchronized (this) { // Synchronizes only this block

count++; // Thread-safe

}

}

}

**Key Points About synchronized:**

**-------------------------------------------**

**Lock Object:** For instance methods, the lock is on the current instance (this). For static methods, the lock is on the class object. For synchronized blocks, you can specify any object as the lock.

**Performance Impact:** Synchronization can cause performance overhead because threads must wait for the lock. Use synchronized blocks to minimize the scope of synchronization and improve performance.

**Reentrant Lock:** Java's synchronized mechanism is reentrant, meaning a thread can acquire the same lock multiple times without deadlocking itself.

**Deadlocks:** Improper use of synchronization can lead to deadlocks, where two or more threads are blocked forever, waiting for each other to release locks.

( Using synchronized (whether on a method or a block of code) introduces overhead and can potentially reduce performance. Therefore, it's crucial to use synchronized only when it's absolutely necessary to ensure thread safety and prevent race conditions or other concurrency issues. )

**→** A synchronized method must have an implementation because synchronization is applied to the method's execution. However, an abstract method cannot have an implementation, which makes the **combination of synchronized and abstract illegal in Java.**

**→ The native modifier** is used to indicate that a method is implemented in platform-specific code (e.g., C, C++) using the Java Native Interface (JNI). It allows Java programs to call code written in other languages, typically for performance reasons or to access system-level functionality that is not available in Java.

Key Features of native Methods:

--------------------------------------------

**No Method Body:** A native method does not have a body in Java. It is implemented in a separate platform-specific language (e.g., C/C++). Example: public native void myMethod();

**Platform-Specific Implementation:** The actual implementation of the method is provided in a shared library (e.g., a .dll file on Windows or a .so file on Linux).

**Used with JNI:** The native modifier is closely associated with the Java Native Interface (JNI), which provides a bridge between Java and native code.

**Performance:** native methods are often used for performance-critical tasks or to interact with low-level system APIs.

Example of native Method:

------------------------------------

public class MyClass {

static {

System.loadLibrary("mylib"); // Load the shared library

}

// Declare the native method

public native void myMethod();

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.myMethod(); // Call the native method

}

}

**C Code (Implementation):**

**------------------------------------**

#include "MyClass.h"

#include <stdio.h>

JNIEXPORT void JNICALL Java\_MyClass\_myMethod(JNIEnv \*env, jobject obj) {

printf("Hello from native code!\n");

} // Output: Hello from native code!

**Key Points About native Methods:**

**----------------------------------------------**

**Platform Dependency:** The shared library is platform-specific, so you need to compile it separately for each platform (e.g., Windows, Linux, macOS).

**Use with Caution:** Using native methods can make your code less portable and harder to maintain. Use them only when necessary.

**When to Use native Methods:** To interact with system-level APIs or hardware that is not accessible from Java. To reuse existing legacy code written in C/C++. For performance-critical tasks where Java's performance is insufficient.

**Steps to Use native Methods:**

**----------------------------------------**

**Declare the native Method:** Declare the method in a Java class using the native keyword. Example:

public class MyClass {

public native void myMethod();

}

**Generate a Header File:** Use the javac and javah tools to generate a C/C++ header file for the native method. Example:

javac MyClass.java

javah -jni MyClass // This generates a header file (e.g., MyClass.h) with the native method signature.

**Implement the Native Method:** Write the implementation of the native method in C/C++ using the generated header file. Example (in C):

#include "MyClass.h"

#include <stdio.h>

JNIEXPORT void JNICALL Java\_MyClass\_myMethod(JNIEnv \*env, jobject obj) {

printf("Hello from native code!\n");

}

**Compile the Native Code:** Compile the C/C++ code into a shared library. Example (on Linux):

gcc -shared -o libmylib.so -I${JAVA\_HOME}/include -I${JAVA\_HOME}/include/linux MyClass.c

**Load the Shared Library in Java:** Use the System.loadLibrary() method to load the shared library in your Java program. Example:

public class MyClass {

static {

System.loadLibrary("mylib"); // Load the shared library

}

public native void myMethod();

public static void main(String[] args) {

MyClass obj = new MyClass();

obj.myMethod(); // Call the native method

}

}

**→** Declaring a native method as strictfp is illegal in Java and will result in a compile-time error. The problem is that you have no control over how the native code implements the floating-point operations. The native code might not adhere to the IEEE 754 standard at all. You're asking the compiler to enforce strict floating-point behavior on code that's outside the compiler's control. The compiler recognizes this and disallows **the combination of native and strictfp.**

**transient Modifier in Java:**

**------------------------------------**

The transient modifier is used to indicate that a variable should not be serialized when the object containing it is serialized. In other words, the value of a transient variable is not saved when the object is written to a file or sent over a network.

Key Features of transient:

-----------------------------------

**Purpose:** The transient keyword is used to exclude specific variables from the serialization process.

**Serialization:** Serialization is the process of converting an object into a byte stream (e.g., for saving to a file or sending over a network). Deserialization is the reverse process of converting the byte stream back into an object.

**Default Value:** When an object is deserialized, transient variables are initialized to their default values (e.g., null for objects, 0 for integers, false for booleans). **Syntax:** transient DataType variableName;

**Use Case:** Use transient for variables that should not be persisted, such as sensitive data (e.g., passwords) or temporary data (e.g., cached values). For non-serializable objects that cannot be saved (e.g., threads, file handles).

**volatile Modifier in Java:**

**----------------------------------**

The volatile modifier is used to indicate that a variable's value may be modified by multiple threads. It ensures that the value of the variable is always read from and written to the main memory, rather than being cached in a thread's local memory. This guarantees visibility of changes across threads, making it useful in multithreaded environments.

Key Features of volatile:

--------------------------------

**Visibility:** Changes made by one thread to a volatile variable are immediately visible to other threads.

**No Caching:** The value of a volatile variable is always read from and written to the main memory, bypassing thread-local caches.

**Atomicity:** Reads and writes to volatile variables are atomic for individual variables (e.g., volatile int). However, compound operations (e.g., volatileVar++) are not atomic. **Syntax:** volatile DataType variableName;

**Performance:** Using volatile can have a slight performance impact because it prevents thread-local caching and enforces main memory access.

**Use Case:** Use volatile for variables that are shared across threads and are updated by one thread and read by others (e.g.,

flags, status variables).

**→ volatile and final together on a variable is illegal in Java** and results in a compilation error (CE) because: final means the variable cannot be modified after initialization. volatile ensures visibility of changes across threads, which implies the variable can be updated. Since these two modifiers have contradictory behaviors, their combination is not allowed.

**All 12 Java modifiers and where they can be used:**

**--------------------------------------------------------------------**

**→ Outer classes:** public, abstract, final, strictfp and default (package-private)

**→ Inner classes:** public, abstract, final, strictfp, default, static, private, protected

**→ Methods can use all except volatile, transient.**

**→ Variables** can use all except abstract, synchronized, native and strictfp.

**→ Blocks** can only use static and synchronized.

**→ Outer Interface:** public, abstract, strictfp and default (package-private)

**→ Inner Interface:** public, abstract, strictfp, default, static, private, protected

**→ Outer Enum:** public, strictfp and default

**→ Inner Enum:** public, default, static, private, protected, strictfp

**→ Constructor:**public, default, private, protected

**(Enums are essentially classes, and strictfp is a valid modifier for classes. The only allowed access modifier for local variables is final)**

**Interface:**

**-------------**

**An interface in Java** is a reference type that defines a contract for classes to implement. It specifies what methods a class must implement but does not provide the implementation details. Interfaces are used to achieve abstraction and multiple inheritance in Java.

(**Durga sir’s definition:** Any service requirement specification or any contract between client and service provider is nothing but interface.)

Key Features of Interfaces:

------------------------------------

**Abstract Methods:** Interfaces can declare abstract methods (methods without a body).

**Default Methods:** Introduced in Java 8, default methods provide a default implementation for methods in an interface.

**Static Methods:** Interfaces can have static methods with a body. Example:

**Constants:** Interfaces can define constants (public, static, and final fields).

**Multiple Inheritance:** A class can implement multiple interfaces, allowing it to inherit behavior from multiple sources. Syntax of an Interface:

interface InterfaceName {

// Abstract methods

void method1();

// Default methods

default void method2() {

// Default implementation

}

// Static methods

static void method3() {

// Static implementation

}

// Constants

int MY\_CONSTANT = 10;

}

**Implementing an Interface:** A class uses the implements keyword to implement an interface. It must provide

implementations for all abstract methods in the interface; otherwise, it must be declared as an abstract class, allowing its subclasses to implement the remaining methods.

**Default Methods in Interfaces:** Default methods allow you to add new methods to an interface without breaking existing implementations. Example:

interface Vehicle {

void start();

void stop();

default void honk() {

System.out.println("Honking!");

}

}

class Car implements Vehicle {

@Override

public void start() {

System.out.println("Car started.");

}

@Override

public void stop() {

System.out.println("Car stopped.");

}

}

public class Main {

public static void main(String[] args) {

Car car = new Car();

car.start(); // Output: Car started.

car.honk(); // Output: Honking!

}

}

**Static Methods in Interfaces:** Static methods in interfaces are called using the interface name and cannot be overridden by implementing classes. Example:

interface Vehicle {

static void service() {

System.out.println("Vehicle is being serviced.");

}

}

public class Main {

public static void main(String[] args) {

Vehicle.service(); // Output: Vehicle is being serviced.

}

}

Key Points About Interfaces:

--------------------------------------

**Cannot Be Instantiated:** You cannot create an object of an interface. Example: Vehicle vehicle = new Vehicle(); // Error: Cannot instantiate an interface

**No Constructors:** Interfaces cannot have constructors.

**All Methods Are Public:** By default, all methods in an interface are **public and abstract.**

**All Fields Are Constants:** By default, all fields in an interface are **public, static, and final.**

**Functional Interfaces:** An interface with exactly one abstract method is called a functional interface.

Example: @FunctionalInterface

interface Runnable {

void run();

}

**When to Use Interfaces:** To define a contract for classes. To achieve abstraction. To support multiple inheritance. To define APIs for libraries or frameworks.

**Example of a Real-World Interface:** The Comparable interface in Java is used to define a natural ordering for objects.

class Student implements Comparable<Student> {

String name;

int age;

public Student(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public int compareTo(Student other) {

return this.age - other.age;

}

}

public class Main {

public static void main(String[] args) {

Student s1 = new Student("Alice", 20);

Student s2 = new Student("Bob", 18);

System.out.println(s1.compareTo(s2)); // Output: 2 (Alice is older than Bob)

}

}

**→** All methods declared in an interface are **implicitly public and abstract,** even if you don't explicitly write these keywords. **Before Java 9,** these were the only allowed modifiers. When a class implements an interface, **it must provide a public implementation** for all the methods declared in the interface. If you try to implement an interface method with any access modifier other than public (like protected, private, or default), you will get a compile-time error. The compiler requires that the implementing method have at least the same level of visibility as the interface method (which is implicitly public). Since you can't reduce visibility when implementing an interface, you must use public.

**→** **A class can extend (inherit from) only one other class at a time.** Java does not support multiple inheritance of classes. An interface can extend any number of other interfaces simultaneously. This is allowed because interfaces only declare methods (contracts), not implementations. A class can implement any number of interfaces simultaneously. The class is then obligated to provide concrete implementations for all the methods declared in all the interfaces it implements.

**→** A class can both extend one class and implement multiple interfaces simultaneously. This is a common and powerful pattern in Java. **The extends clause must come before the implements clause.**

Example: class Novel extends Book implements Printable, Readable {

// Extends one class, implements two interfaces

}

**→** Interface methods are implicitly public to ensure they are accessible to all implementing classes and abstract to enforce method implementation by those classes. You can explicitly specify the access modifier and abstraction in an interface method, like: **public abstract void method1();** However, since interface methods are implicitly public and abstract (except for default and static methods), explicitly mentioning them is optional.

**→** Since interface methods are implicitly public and abstract, the following modifiers are illegal when used with interface methods: final, static, strictfp, private, protected, native, synchronized

**→** **Interfaces can have static methods with a body. (introduced in Java 8)**

Example: static void myStaticMethod() {

System.out.println("Static method in interface");

}

(If you explicitly declare a method as static in an interface, it is implicitly public, even if you do not specify the public keyword. Since Java 8, static methods in interfaces must always be public, and using any other access modifier (like private, protected or package-private) will result in a compilation error)

**→** An interface can contain variables, and their main purpose is to define requirement-level constants. Key Points about Interface Variables:

**Implicit Modifiers:** All variables in an interface are implicitly public, static, and final.

**Initialization Requirement:** They must be initialized at the time of declaration.

**Cannot Be Modified:** Since they are final, their values cannot be changed once assigned. Example:

interface Config {

int MAX\_USERS = 100; // Implicitly public, static, and final

String APP\_NAME = "MyApp";

}

class Test implements Config {

public static void main(String[] args) {

System.out.println(MAX\_USERS); // Accessing directly

System.out.println(Config.APP\_NAME); // Accessing with interface name

}

} // This approach ensures that constant values remain consistent across all implementing classes.

**→** Here's why variables declared in interfaces are **implicitly public, static, and final:**

**public:** The variables declared in an interface are meant to be part of that contract, accessible to any class that implements the interface. Therefore, they must be public.

**static:** Interface variables are implicitly static. They belong to the interface itself, not to instances of classes that implement the interface. There's only one copy of each interface variable, shared by all implementing classes.

**final:** Interface variables are implicitly final. They represent constants. Their values cannot be changed after they are initialized. This is essential because interface variables are part of the contract, and changing their values would violate that contract and could lead to unpredictable behavior in the classes that implement the interface.

**→** You can explicitly declare a variable as public, static, and final in an interface, but it is redundant.

Example: interface Constants {

int MAX\_USERS = 100; // Implicitly public, static, and final

public static final int TIMEOUT = 30; // Explicit declaration (redundant)

}

( Both MAX\_USERS and TIMEOUT behave the same way. The explicit declaration makes no difference in functionality.)

**→ It is illegal to use the following modifiers with interface variables:** private, protected, transient, volatile

**→** If two interfaces have a method with the same signature and same return type, and a class implements both interfaces, then the class only needs to implement that method once. Example:

interface A {

void show();

}

interface B {

void show();

}

class Test implements A, B {

public void show() { // Single implementation for both interfaces

System.out.println("Implementation of show()");

}

public static void main(String[] args) {

Test obj = new Test();

obj.show(); // Output: Implementation of show()

}

} // Since both interfaces define the same method, the implementing class provides one implementation that satisfies both interfaces.

**→** If two interfaces contain a method with **the same signature but different return types,** it is impossible to implement both interfaces simultaneously unless the return types are covariant. Allowed When Return Types Are Covariant:

interface A {

Number show();

}

interface B {

Integer show();

}

class Test implements A, B {

public Integer show() { // Allowed: Integer is a subtype of Number (covariant)

return 10;

}

}

// Since Integer is a subtype of Number, the method show() in Test satisfies both interfaces without conflict. **Thus, if return types are not covariant, implementing both interfaces is impossible.**

**→** Two interfaces can declare variables with the same name, which can lead to naming conflicts when a class implements both interfaces. However, this conflict can be resolved by **using the interface name** to qualify the variable, thereby disambiguating the reference.

**→ A marker interface** is a special type of interface in Java that does not declare any methods or fields. It is used to mark or tag a class with a specific capability or behavior. The presence of the marker interface indicates that the class has a certain property or can be treated in a special way by the JVM or other frameworks.

Key Features of Marker Interfaces:

----------------------------------------------

**No Methods:** A marker interface does not contain any methods or fields. It is empty and serves only as a tag.

**Runtime Behavior:** The JVM or frameworks check for the presence of the marker interface at runtime to enable specific

behavior. **Examples in Java:**

**Serializable:** The Serializable interface is used to indicate that an object can be serialized (converted into a byte stream).

**Cloneable:** The Cloneable interface indicates that an object can be cloned using the clone() method.

**Remote:** Used in RMI (Remote Method Invocation) to mark remote objects.

Why Use Marker Interfaces?

---------------------------------------

**Simplicity:** They provide a simple way to add metadata or tags to a class.

**Runtime Checks:** Frameworks or the JVM can use reflection to check for the presence of a marker interface and enable specific behavior.

**Backward Compatibility:** They are often used to maintain compatibility with older code or frameworks.

Marker Interface vs. Annotations:

----------------------------------------------

**Marker Interfaces:** Used in older versions of Java. Provide runtime behavior through reflection.

**Annotations:** Introduced in Java 5. Provide a more flexible and powerful way to add metadata

to classes, methods, or fields.

When to Use Marker Interfaces:

-------------------------------------------

When you need to tag a class with specific behavior or metadata. When working with frameworks or libraries that rely on

marker interfaces (e.g., Serializable, Cloneable). For backward compatibility with older code.

(Internally, **the Java Virtual Machine (JVM) provides the necessary functionality** for marker interfaces, also known as tag interfaces. Marker interfaces, such as Serializable, Cloneable, and Comparable, do not have any methods or variables, but instead serve as a signal to the JVM to provide special treatment or behavior. By leveraging marker interfaces, Java programmers can write simpler, more concise code, while the JVM handles the underlying complexity.)

**→ An adapter class** in Java is a class that provides an empty (default) implementation of all methods in an interface, allowing subclasses to override only the methods they need. This is useful when an interface has multiple methods, but a class implementing it doesn't need to provide implementations for all of them.

**Key Points about Adapter Class:** It is an **abstract class** that implements an interface and provides default (empty) implementations of its methods. It helps reduce boilerplate code by allowing subclasses to override only the required methods. (Mainly used for listener interfaces in event handling (e.g., MouseAdapter, KeyAdapter, WindowAdapter))

**Ex: GenericServlet** acts much like an adapter class for the Servlet interface. It implements the interface and provides default implementations for several lifecycle methods—such as init(), getServletConfig(), getServletInfo(), and destroy()—while leaving the service() method abstract. This design forces subclasses to implement the service() method, ensuring that the servlet's request-handling logic is defined by the developer.

**→** When designing a class hierarchy, the choice between an interface, abstract class, and concrete class depends on the level of implementation required:

**Interface:** Choose when you only know the service requirements, but not the implementation details. Interfaces define a contract or a set of methods that must be implemented, without providing any implementation.

**Abstract Class**: Choose when you want to provide partial implementation, i.e., some methods are implemented, while others are left abstract. Abstract classes provide a basic implementation that can be shared by subclasses, while still allowing for customization.

**Concrete Class:** Choose when you want to provide a full implementation, i.e., all methods are implemented, and the class can be instantiated directly. Concrete classes provide a complete implementation of a class, without any abstract methods.

**→** Abstract class methods can have various access modifiers, Non-abstract methods (i.e., methods with an implementation) in an abstract class can have any access modifier, including private.

**→** Both interfaces and abstract classes are used to achieve abstraction in Java, but they serve different purposes and have distinct characteristics. Here's a detailed comparison:

**Interface**  **Abstract Class**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Definition:**

An interface is a reference type that An abstract class is a class that cannot be defines a contract for classes. instantiated and may contain abstract methods.

It specifies what methods a class must implement. It can provide partial implementation and force subclasses to implement the rest.

**Methods:**

All methods are public and abstract Can have abstract methods (without a body) and

by default (before Java 8). concrete methods (with a body).

From Java 8, interfaces can have Can have any type of method (abstract, concrete,

default and static methods. static, final, etc.)

From Java 9, interfaces can have private methods. Can have private, protected, and public methods.

**Variables:**

All variables are public, static, final Can have instance variables, static variables,

by default (constants). and constants.

Cannot have instance variables. Can have variables with any access modifier (private, protected, public).

**Inheritance:**

A class can implement multiple interfaces A class can extend only one abstract class

(supports multiple inheritance). (does not support multiple inheritance).

An interface can extend multiple interfaces. An abstract class can extend only one class (abstract or concrete).

**Constructors:**

Cannot have constructors. Can have constructors. **Constructors**

**are used to initialize the state of the subclass.**

**Instantiation:**

Cannot be instantiated. Cannot be instantiated. Can be used as a reference type for subclass objects.

**Use Cases:**

Used to define a contract or API for classes. Used to provide a common base class for Suitable for multiple inheritance and loose coupling. subclasses. Suitable for code reuse and shared functionality.

**→** An abstract class can have a constructor because it is still a class, and **constructors are used to initialize fields of the class when an object is created.** Even though an abstract class cannot be instantiated directly, **its constructor is called when a subclass is instantiated,** ensuring proper initialization of inherited fields. (Constructors in abstract classes are used to facilitate code reusability)

On the other hand, an interface does not have a constructor because: **Interfaces do not have instance variables to initialize.** They define behavior rather than state. They are meant to be implemented by classes, which provide their own constructors for initialization. (Thus, abstract classes support constructors for inheritance-based initialization, while interfaces do not require them)

**→** We can replace an interface with an abstract class containing only abstract methods, but it is not recommended because:

**Loss of Multiple Inheritance→** Since Java allows a class to implement multiple interfaces but can extend only one class, using an abstract class restricts flexibility.

**Higher Object Creation Cost** **→** Abstract classes can have constructors, instance variables, and instance methods, leading to unnecessary memory consumption and increasing object creation overhead.

**Not a True Contract** **→** Interfaces define a strict contract that multiple classes can implement independently, while abstract classes suggest a common hierarchy, which may not always be needed.

(Thus, interfaces are preferred when defining a contract for multiple classes, while abstract classes are used when some common implementation is required.)

**Object Oriented Programming**

**====================================================================================**

**→ Data Hiding** is a principle of object-oriented programming (OOP) that restricts direct access to an object's data from outside the class. It ensures that sensitive data is protected and can only be accessed or modified **through well-defined**

**methods.**

**How Data Hiding Works?**

**---------------------------------**

**Using private access modifier:** Instance variables are declared **as private** to prevent direct access from outside the class.

**Providing public getter and setter methods:** Controlled access is given through methods to retrieve (get) or update (set) the values.

**Encapsulation:** Data hiding is a key part of encapsulation, where implementation details are hidden, and only necessary information is exposed.

Benefits of Data Hiding:

**---------------------------------**

**Security** **→**Prevents accidental or malicious modifications.

**Encapsulation** **→** Maintains data integrity by restricting direct access.

**Flexibility** **→** Implementation details can change without affecting external code.

**Code Maintainability** **→** Makes debugging and modifying code easier.

**Example:**

**-------------**

class BankAccount {

private double balance; // Data is hidden from outside

public BankAccount(double balance) {

this.balance = balance;

}

// Getter method to access balance

public double getBalance() {

return balance;

}

// Setter method to update balance with validation

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

} else {

System.out.println("Invalid deposit amount");

}

}

}

**Abstraction in Java:**

**--------------------------**

Abstraction is one of the four fundamental principles of Object-Oriented Programming (OOP). It refers to the process of hiding the implementation details and showing only the essential features of an object. In simpler terms, abstraction allows you to focus on what an object does rather than how it does it.

Key Concepts of Abstraction:

--------------------------------------

**Hide Complexity:** Abstraction hides the internal complexity of a system and provides a simplified interface to the user.

**Focus on Essentials:** It allows you to focus on the high-level functionality of an object without worrying about the low-level implementation details.

Achieved Through:

--------------------------

**Abstract Classes:** Classes that cannot be instantiated and may contain abstract methods.

**Interfaces:** Contracts that define a set of methods without providing implementation.

Why Use Abstraction?

------------------------------

**Simplifies Code:** Abstraction reduces complexity by hiding unnecessary details.

**Improves Maintainability:** Changes to the internal implementation do not affect the external interface.

**Enhances Reusability:** Abstract classes and interfaces can be reused across multiple classes.

**Promotes Flexibility:** Abstraction allows you to change the implementation without affecting the overall system.

How Abstraction is Achieved in Java:

-------------------------------------------------

**Abstract Classes:** An abstract class is a class that cannot be instantiated. It can contain abstract methods (methods without a body) and concrete methods (methods with a body). Subclasses must provide implementations for the abstract methods. **Example:**

abstract class Animal {

// Abstract method (no body)

abstract void makeSound();

// Concrete method

void sleep() {

System.out.println("Animal is sleeping.");

}

}

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks.");

}

}

**Interfaces:** An interface is a reference type that defines a contract for classes. It contains abstract methods (before Java 8) and can also have default and static methods (from Java 8 onwards). Classes implement interfaces and provide implementations for the abstract methods. **Example:**

interface Vehicle {

void start(); // Abstract method

void stop(); // Abstract method

}

class Car implements Vehicle {

@Override

public void start() {

System.out.println("Car started.");

}

@Override

public void stop() {

System.out.println("Car stopped.");

}

}

**Encapsulation in Java:**

**------------------------------**

Encapsulation is one of the four fundamental principles of Object-Oriented Programming (OOP). It refers to the concept of bundling data (attributes) and methods (behavior) that operate on the data into a single unit, typically a class. Additionally, encapsulation restricts direct access to some of an object's components, which is achieved using access modifiers like private, protected, and public.

Key Concepts of Encapsulation:

------------------------------------------

**Data Hiding:** Encapsulation hides the internal state of an object and only exposes a controlled interface to interact with it.

**Access Control:** Access modifiers (private, protected, public) are used to control the visibility of data and methods.

**Bundling:** Data (fields) and methods (functions) that operate on the data are bundled together in a class.

Why Use Encapsulation?

---------------------------------

**Data Protection:** Prevents unauthorized access and modification of data.

**Improved Maintainability:** Changes to the internal implementation do not affect the external interface.

**Flexibility:** Allows you to change the internal implementation without affecting other parts of the code.

**Reusability:** Encapsulated classes can be reused in different parts of the program.

How Encapsulation is Achieved in Java:

----------------------------------------------------

**Private Fields:** Declare fields as private to restrict direct access from outside the class.

**Public Getter and Setter Methods:** Provide public methods (getters and setters) to access and modify the private fields. Example:

class Person {

// Private fields

private String name;

private int age;

// Public getter for name

public String getName() {

return name;

}

// Public setter for name

public void setName(String name) {

this.name = name;

}

// Public getter for age

public int getAge() {

return age;

}

// Public setter for age

public void setAge(int age) {

if (age > 0) { // Validation

this.age = age;

} else {

System.out.println("Invalid age.");

}

}

}

**Aspect Encapsulation Abstraction**

**-------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Focus Bundling data and methods into a single unit. Hiding implementation details.

Purpose Protects data from external access. Simplifies complexity.

Achieved Through Access modifiers (private, protected, public). Abstract classes, interfaces.

**→** A class is considered **tightly encapsulated** if all its instance variables are **declared as private,** regardless of whether getter and setter methods exist or their access levels. The key requirement is that no variable should have a direct access modifier other than private.

Example of a Tightly Encapsulated Class:

------------------------------------------------------

class Person {

private String name; // Private variables (tightly encapsulated)

private int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

// Getter method

public String getName() {

return name;

}

// Setter method

public void setName(String name) {

this.name = name;

}

}

( **If a parent class is not tightly encapsulated,** then no child class of that parent can be truly tightly encapsulated, regardless of how its own members are declared.)

**→ The IS-A relationship** represents inheritance or interface implementation in Java. It signifies that a subclass is a specialized version of a superclass or that a class implements an interface.

Types of IS-A Relationships:

-------------------------------------

**Class Inheritance** (extends) **→** A subclass inherits from a parent class.

**Interface Implementation** (implements) **→** A class implements an interface.

( Key Points: Ensures code reusability, Supports polymorphism )

**→** Parent class methods are inherited by the child class → The child class can access both parent and its own methods.

**→** Parent class reference can hold a child class object → But it can only call methods defined in the parent class (unless overridden).

**→** Child class reference cannot hold a parent class object → This would break the inheritance rule. Example:

class Parent {

void parentMethod() {

System.out.println("Parent method");

}

}

class Child extends Parent {

void childMethod() {

System.out.println("Child method");

}

}

public class Test {

public static void main(String[] args) {

Child c = new Child();

c.parentMethod(); // Allowed (inherited from Parent)

c.childMethod(); // Allowed (defined in Child)

Parent p = new Child();

p.parentMethod(); // Allowed (Parent's method is available)

// p.childMethod(); // Compile-time error (Parent reference can't access Child-specific methods)

// Child c2 = new Parent(); // Compile-time error (Parent cannot be assigned to Child reference)

}

}

**→ The Object class is the root of all Java classes,** meaning every class in Java either directly or indirectly extends Object. There are **11** common methods in Object Class, since every Java class inherits from Object, these methods are available in all Java classes by default. Some methods **(equals(), toString(), hashCode())** are commonly overridden in user-defined classes. Methods like wait(), notify(), and notifyAll() are used for thread synchronization.

**→** Java does not support multiple inheritance of state, meaning a Java class cannot extend more than one parent class at a

time. When a class does not explicitly extend any other class, it automatically becomes a direct child of the Object class. However, if a class extends another class, it becomes an indirect child of Object **through multilevel inheritance.** Thus, multilevel inheritance ensures that all Java classes eventually inherit from Object, either directly or indirectly.

**→ Java does not support multiple inheritance** with classes to avoid ambiguity, complexity, and the diamond problem.

**Reasons:** Diamond Problem: If a class inherits from two classes having the same method, the compiler cannot decide which method to call. Ex:

class A {

void show() { System.out.println("A's show"); }

}

class B {

void show() { System.out.println("B's show"); }

}

class C extends A, B { } // Not allowed in Java

**Ambiguity & Complexity:** When multiple parent classes have methods with the same name, it becomes unclear which method should be executed. It makes code harder to maintain and debug.

**Avoids Confusion in Constructor Chaining:** If multiple parent classes have constructors, resolving constructor calls in the child class would be complicated and error-prone.

**→** Java supports multiple inheritance through interfaces, as interfaces do not have implementation conflicts. Ex:

interface A {

void show();

}

interface B {

void show();

}

class C implements A, B {

public void show() { System.out.println("C's show"); }

}

**→** Thus, Java avoids multiple inheritance with classes but allows it using interfaces to ensure flexibility without ambiguity.

**→ Cyclic inheritance is not allowed in Java, and it is also unnecessary.**

Why is cyclic inheritance not allowed?

---------------------------------------------------

**Logical contradiction:** A class cannot be both a parent and a child of itself.

**Compiler error:** Java does not allow a class to extend itself directly or indirectly, as it leads to an infinite inheritance loop.

**Breaks hierarchical structure:** Java follows a well-structured inheritance model, and cyclic dependencies disrupt the class hierarchy.

Example (Invalid in Java):

----------------------------------

class A extends A { // Compilation error

}

class A extends B { }

class B extends A { } // Compilation error (indirect cyclic dependency)

Why is it not required?

-------------------------------

**No practical use case:** Cyclic inheritance does not provide any meaningful benefit.

**Causes infinite loops & stack overflow:** If cyclic inheritance were allowed, method calls would create an infinite recursive

loop.

**→ The HAS-A relationship** is indeed also known as aggregation or composition but there is no specific keyword to define the HAS-A relationship. Typically implemented using the new keyword or dependency injection. Used for code reusability by allowing objects to be composed of other objects instead of inheriting from them. Composition provides better flexibility compared to inheritance because it allows dynamic object relationships. Example of HAS-A Relationship (Composition):

class Engine {

void start() {

System.out.println("Engine starting...");

}

}

class Car {

private Engine engine = new Engine(); // HAS-A relationship

void startCar() {

engine.start();

System.out.println("Car started...");

}

}

**Why Prefer Composition Over Inheritance?**

**-----------------------------------------------------------**

More flexible (allows changing behavior at runtime). Encapsulation is stronger (child class does not depend on parent implementation). Avoids tight coupling and diamond problem of multiple inheritance. Thus, HAS-A relationships improve modularity and code reusability.

**→** If a contained object **cannot exist independently without the container object**, then the container and contained objects have a strong association, which is called **composition.**

**Key Points About Composition:**

**-------------------------------------------**

Stronger association than aggregation. Contained objects have no independent lifecycle, if the container is destroyed, the contained object is also destroyed. Implemented using instance variables in the container class. Example:

class Heart {

void beat() {

System.out.println("Heart is beating...");

}

}

class Human {

private final Heart heart; // Strong association (Composition)

public Human() {

this.heart = new Heart(); // Created inside Human

}

void live() {

heart.beat();

System.out.println("Human is alive...");

}

}

**→ Aggregation is a weak association** between a container (whole) and contained (part) objects, where the contained object can exist independently of the container.

Key Points About Aggregation:

-----------------------------------------

Weaker association than composition. Contained objects have an independent lifecycle, they can exist even if the container is destroyed. Implemented using instance variables with references passed from outside (not created inside the container). This loose coupling and independent existence of the contained object make it an aggregation relationship. Example:

class Engine {

void start() {

System.out.println("Engine is starting...");

}

}

class Car {

private final Engine engine; // Weak association (Aggregation)

public Car(Engine engine) { // Engine is passed from outside

this.engine = engine;

}

void drive() {

engine.start();

System.out.println("Car is moving...");

}

}

( **IS-A** (Inheritance) Relationship **→** Use when you need **all functionalities** of the parent class.

**HAS-A** (Composition/Aggregation) Relationship **→** Use when you need **only specific functionalities** without inheriting everything. This approach improves code reusability and flexibility while reducing unnecessary dependencies.)

**→** In Java, a method signature consists of: **Method Name + Parameter Types (in order)**

**It does not include:** Return type, Access modifiers (public, private, etc.), throws clause

For example, both methods have the same signature despite different return types:

int add(int a, int b) { return a + b; }

double add(int a, int b) { return a + b; } // Compilation Error (same signature)

( The Java compiler uses method signatures to resolve method calls, a process known as method overloading resolution or method invocation resolution. Within a class, two methods with the same signature are not allowed. This is because method signatures must be unique within a class to avoid ambiguity and ensure that the correct method is invoked.)

**Overloading in Java:**

**---------------------------**

Overloading is a feature in Java that allows a class to have multiple methods with the same name but with different parameters. It is a way to achieve compile-time polymorphism. Overloading is used to increase the readability and reusability of the code.

Key Features of Overloading:

---------------------------------------

**Same Method Name:** Methods must have the same name.

**Different Parameters:** Methods must have different parameter lists (different number of parameters, different types of parameters, or both).

**Return Type:** The return type of the methods can be the same or different. However, the return type alone is not sufficient to differentiate overloaded methods.

**Compile-Time Polymorphism:** The method to be called is determined at compile time based on the method signature.

Types of Overloading:

------------------------------

**Method Overloading:** Multiple methods in the same class with the same name but different parameters. Example:

class Calculator {

// Method to add two integers

int add(int a, int b) {

return a + b;

}

// Overloaded method to add three integers

int add(int a, int b, int c) {

return a + b + c;

}

// Overloaded method to add two double values

double add(double a, double b) {

return a + b;

}

}

**Constructor Overloading:** Multiple constructors in the same class with the same name but different parameters. Example:

class Student {

String name;

int age;

// Constructor with no parameters

Student() {

this.name = "Unknown";

this.age = 0;

}

// Constructor with one parameter

Student(String name) {

this.name = name;

this.age = 0;

}

// Constructor with two parameters

Student(String name, int age) {

this.name = name;

this.age = age;

}

void display() {

System.out.println("Name: " + name + ", Age: " + age);

}

}

**Rules for Overloading:**

**-------------------------------**

Parameter List Must Differ. Return Type Does Not Matter.

**Access Modifiers Can Vary:** Overloaded methods can have different access modifiers (e.g., public, private, protected).

**Throws Clause Can Vary:** Overloaded methods can have different exception lists.

Key Points About Overloading:

-----------------------------------------

**Improves Readability:** Overloading allows you to use the same method name for similar operations, making the code more

readable.

**Increases Reusability:** Overloaded methods can be reused with different parameters.

( **In method overloading**, the method to be invoked is determined by the compiler at **compile-time**, based on the reference type and the number and types of arguments passed. This process is known as **compile-time polymorphism, static polymorphism, or early binding.**)

**→** While resolving overloaded methods, if an exact match is not found, Java follows automatic type promotion to find a compatible method.

**→ Overloading can happen in the same class or between a parent and child class.**

**Method Resolution Process:**

**---------------------------------------**

**Exact match → If found, it is selected.**

Widening conversion (Implicit Promotion) → Converts smaller types to larger types (byte → short → int → long → float → double)

Boxing/Unboxing → Converts primitive types to wrapper classes (int → Integer etc.)

Varargs method → **Considered last (Least priority)** This means that a vararg method will only be considered if no other method matches the method invocation, making it a fallback option.

Compilation Error (CE) → If no method is found. Example:

class Test {

void m(int x) { System.out.println("int"); }

void m(double x) { System.out.println("double"); }

public static void main(String[] args) {

Test t = new Test();

t.m(10); // Exact match → Calls m(int)

t.m(10.5f); // float → double (widening) → Calls m(double)

t.m(10L); // long → double (widening) → Calls m(double)

}

} // If no method is found after all promotions, a Compile-Time Error (CE) occurs.

**→** While resolving overloaded methods, the compiler gives **precedence to the most specific method.**

**Rule:** If multiple overloaded methods exist, the method with the **child type argument is preferred over the parent** type. If a method with an exact match is found, it is selected. If multiple methods match, the most specific one (i.e., the method with the subclass type parameter) is chosen. If ambiguity occurs (e.g., two unrelated classes), a Compile-Time Error (CE) happens.

Example:

class Test {

void m1(Object o) { System.out.println("Object method"); }

void m1(String s) { System.out.println("String method"); }

public static void main(String[] args) {

Test t = new Test();

t.m1(null); // Calls String method (more specific than Object)

}

} // Output: String method

Another Example (Ambiguity Case):

-----------------------------------------------

class Test {

void m1(String s) { System.out.println("String method"); }

void m1(StringBuilder sb) { System.out.println("StringBuilder method"); }

public static void main(String[] args) {

Test t = new Test();

t.m1(null); // CE: reference to m1() is ambiguous

}

}

**Reason:** Both String and StringBuilder are not related (not parent-child), so the compiler cannot decide which method to call, resulting in a Compile-Time Error (CE).

An ambiguity only occurs if neither method matches exactly and implicit type promotion causes confusion. For example, if we had:

class Test {

void m1(double num1, int num2) { System.out.println("double, int method"); }

void m1(int num1, double num2) { System.out.println("int, double method"); }

public static void main(String[] args) {

Test t = new Test();

t.m1(10, 10); // CE: reference to m1() is ambiguous

}

}

( Here, both m1(double, int) and m1(int, double) are valid promotions for (10,10), causing ambiguity.)

**→ In method overloading, method resolution is always done at compile-time** based on the reference type. The runtime object does not affect method selection. Example:

class Parent {

void m1() { System.out.println("Parent method"); }

}

class Child extends Parent {

void m1(int x) { System.out.println("Child method"); }

}

public class Test {

public static void main(String[] args) {

Parent p = new Child();

p.m1(); // Calls Parent's m1() (Compile-time resolution)

// p.m1(10); // CE: cannot find symbol (because Parent reference doesn't have m1(int))

}

}

**( Explanation:** The compiler checks the reference type (Parent), not the runtime object (Child). Since Parent does not have m1(int), calling p.m1(10); gives a compilation error. Overloading does not depend on runtime polymorphism, unlike method overriding.**)**

**Overriding in Java:**

**-------------------------**

Overriding is a feature in Java that allows a subclass to provide a specific implementation of a method that is already defined in its superclass. It is used to achieve runtime polymorphism. Overriding enables a subclass to redefine or extend the behavior of a method inherited from its superclass.

Key Points About Overriding:

---------------------------------------

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

**Runtime Polymorphism:** The method to be called is determined at runtime based on the object's type.

**Access Modifier:** The access modifier of the overriding method **cannot be more restrictive** than the overridden method.

**Exception Handling:** The overriding method can throw the same, subclass, or no exception compared to the overridden method. It cannot throw a broader exception.

**Return Type Must Be Compatible:** The return type of the overriding method must be the same or a subtype of the return type of the overridden method.

**Cannot Override Static, Final, or Private Methods:** Static methods are bound to the class, not the object, so they cannot be

overridden. **Final methods cannot be overridden.** Private methods are not accessible in the subclass, so they cannot be overridden.

**Use @Override Annotation:** The @Override annotation is used to indicate that a method is intended to override a method in the superclass. It helps catch errors at compile time. Example of Overriding:

Superclass:

---------------

class Animal {

void makeSound() {

System.out.println("Animal makes a sound.");

}

}

Subclass:

-------------

class Dog extends Animal {

@Override

void makeSound() {

System.out.println("Dog barks.");

}

}

( In method overriding, the method to be invoked is determined **by the JVM at runtime**, based on the actual object type, rather than the reference type. This process is known as runtime polymorphism, dynamic polymorphism, or late binding.)

**→ Covariant return type is only applicable for object types, not for primitive types.**

**Explanation:** In method overriding, the child class can change the return type of the overridden method only if the return type is a subclass of the parent’s return type (i.e., co-variant return type). This does not apply to primitive types, as they do not follow inheritance. Example:

class Parent {

Object m1() { return "Parent"; }

}

class Child extends Parent {

@Override

String m1() { return "Child"; } // Valid (String is a subclass of Object)

}

Example (Invalid - Primitive Types, Causes CE):

class Parent {

int m1() { return 10; }

}

class Child extends Parent {

@Override

double m1() { return 10.5; } // CE: return type is incompatible

}

**→ We can override a non-abstract method as abstract,** but only when the subclass itself is declared abstract.

**Explanation:** A concrete (non-abstract) method in the parent class can be overridden in the child class. If the child class does not provide an implementation, it must be declared abstract. This forces the next concrete subclass to implement the method. Example:

class Parent {

void display() { // Concrete method

System.out.println("Parent display method");

}

}

abstract class Child extends Parent {

abstract void display(); // Overridden as abstract

}

( Here, Child overrides display() but does not provide an implementation, so it must be declared abstract. The main advantage of overriding a non-abstract method as abstract is to stop the parent’s implementation from propagating to further subclasses. This forces the next level of child classes to provide their own specific implementation.)

**→** In method overriding, the following modifiers do not impose any restrictions, meaning you can freely use them or change them in the overridden method: **synchronized, native, strictfp, abstract** ( Can be added or removed in the overridden method.)

**→** In method overriding, the overriding method (child class) cannot have a more restrictive access modifier than the overridden method (parent class). Access Modifier Order (Least to Most Restrictive)

**public → protected → default (package-private) → private**

public method in the parent class → Must remain public in the child.

protected method in the parent class → Can be protected or public, but not default or private.

default (package-private) method in the parent class → Can be default, protected, or public

**→ private methods are not inherited, so overriding does not apply. Ex:**

class Parent {

private void method() {} // Not inherited, no overriding

}

class Child extends Parent {

void method() {} // This is a new method, not an override

}

No restriction **(it’s a new method, not an override)**

**Exception Handling Rules in Method Overriding:**

**-----------------------------------------------------------------**

When a child class method overrides a parent class method, the checked exceptions it throws must follow specific rules. However, unchecked exceptions have no such restrictions.

Rules for Checked Exceptions:

-----------------------------------------

The child method can throw the same checked exception as the parent method. The child method can throw a subclass of the checked exception declared in the parent method. Ex:

class Parent {

void method() throws Exception {} // Parent method throws Exception

}

class Child extends Parent {

void method() throws IOException {} // Allowed (IOException is a subclass of Exception)

}

The child method cannot throw a checked exception that is not declared in the parent method. The child method cannot throw a broader checked exception than the parent.

class Parent {

void method() throws IOException {} // Parent method throws IOException

}

class Child extends Parent {

void method() throws Exception {} // Compilation Error (Exception is broader than IOException)

}

Rules for Unchecked Exceptions:

--------------------------------------------

**No restrictions →** The child method can throw any unchecked exception, even if the parent method does not throw any.

Example:

class Parent {

void method() {} // No exception declared

}

class Child extends Parent {

void method() throws ArithmeticException, NullPointerException {} // Allowed

}

**→ A static method** cannot be overridden as a non-static method. A non-static method cannot be overridden as a static

method. Ex:

class Parent {

void method() {} // Non-static method in parent

}

class Child extends Parent {

static void method() {} // Compilation Error: Cannot override instance method as static

}

**→** Static methods are inherited but not overridden (Method Hiding). If a child class defines a static method with the same signature as a static method in the parent class, **it is called method hiding**, not overriding.

**Key Differences between Method Overriding and Method Hiding:**

**Aspect Method Overriding Method Hiding**

**-------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Applicable To Instance methods. Static methods.

Polymorphism Runtime polymorphism. No runtime polymorphism.

Binding Dynamic binding Static binding

(resolved at runtime). (resolved at compile time).

Method Resolution Based on the object's type. Based on the reference type.

Use of @Override annotation @Override is used. @Override is not used.

Example Overriding an instance Hiding a static method in a subclass. method in a subclass.

**( Note:** Method Overriding means the parent's **method is completely replaced** and cannot be accessed via a child object.

But in Method Hiding, **both methods exist separately**, and we can access them based on the reference type.**)**

**→** We can override a vararg method only with another vararg method that has the same method name, return type, and

parameter types.

**→** In Java**,** variable resolution is always based on the **reference type at compile time**, regardless of whether the variable is

static or non-static. **Overriding applies only to methods, not to variables.**

class Parent {

int x = 10;

static int y = 20;

}

class Child extends Parent {

int x = 30; // Hides Parent's x

static int y = 40; // Hides Parent's y

}

public class Test {

public static void main(String[] args) {

Parent p = new Parent();

System.out.println(p.x); // Output: 10

System.out.println(p.y); // Output: 20

Child c = new Child();

System.out.println(c.x); // Output: 30

System.out.println(c.y); // Output: 40

Parent p1 = new Child(); // Parent reference holding Child object

System.out.println(p1.x); // Output: 10

System.out.println(p1.y); // Output: 20

}

}

**( Key Takeaways:** Variable resolution happens at compile-time based on the reference type. Thus, variables cannot be

overridden, they can only be hidden when redefined in a subclass.**)**

**Differences Between Overloading and Overriding:**

**Overloading Overriding**

**------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Definition:**

Overloading occurs when a class has multiple Overriding occurs when a subclass provides a

methods with the same name already specific implementation of a method

but different parameters. that is defined in its superclass.

**Applicable To:**

Can occur in the same class or between Occurs only in a subclass (child class).

a parent and child class.

**Method Signature:**

Methods must have the same name but Methods must have the same name,

different parameter lists return type, and parameter list.

(number, type, or order of parameters).

**Polymorphism:**

Supports compile-time polymorphism Supports runtime polymorphism

(the method to be called is determined at compile time). (the method to be called is determined at runtime).

**Binding:**

Uses static binding (the method is bound at compile time). Uses dynamic binding (the method is bound at runtime).

**Return Type:**

Return type can be the same or different. Return type must be the same or a subtype (covariant return type).

**Access Modifier:**

Access modifiers can be different. The access modifier of the overriding method cannot be more

restrictive than the overridden method.

**Exception Handling:**

Overloaded methods can have The overriding method can throw the same, subclass,

different exception lists. or no exception compared to the overridden method. It cannot throw a broader exception.

**Use of @Override Annotation:**

The @Override annotation is not used. The @Override annotation is used to indicate that a method

is intended to override a method in the superclass.

**→ Polymorphism is the ability** of an object or a method to take on multiple forms, depending on the context in which it is used. This allows objects of different classes to be treated as objects of a common superclass, enabling more flexibility and generic code.

**→** Parent class reference can hold a child class object, but it can only access the methods declared in the parent class. It cannot call child-specific methods unless they are overridden versions of parent methods. Child class reference can access both parent and child methods because it has complete knowledge of both.

**→ Use a parent reference** when you don’t know the exact runtime type of the object or when you want to write generic code that can work with multiple child classes (polymorphism). Use a child reference when you know the exact runtime type and need to access child-specific methods.

**Compile-time Polymorphism:** Method Overloading, Method Hiding

**Runtime Polymorphism:** Method Overriding

**→ Coupling refers to the degree of interdependence** or connection between two or more components, modules, or classes in a software system. It measures how closely connected they are and how much one component affects or relies on another.

Tight coupling (high dependency) can make software harder to maintain, modify, and extend, while loose coupling (low dependency) promotes modularity, flexibility, and scalability. **Loose coupling is a good programming practice.**

**→** Tightly coupling, where different parts of a software system are heavily dependent on each other, has several significant disadvantages:

**Reduced Reusability:** Tightly coupled components are difficult to reuse in other parts of the system or in different projects.

**Decreased Flexibility and Adaptability:** Tightly coupled systems are less flexible and adaptable to changing requirements. It's harder to modify or replace individual components without affecting other parts of the system.

**Increased Maintenance Costs:** Changes in one part of the system can have ripple effects throughout the tightly coupled components. A seemingly small change in one module might require modifications in many other modules.

**→** Minimizing dependencies between components is **the essence of loose coupling.** It's a design principle that aims to make components more independent and less reliant on each other.

**→ Cohesion refers to the degree** to which a component, module, or class has a clear, well-defined, and focused functionality. High cohesion means that a component performs a single, specific task or set of related tasks, making it more modular, maintainable, and reusable.

**→ High cohesion is a good programming practice** as it improves code maintainability, readability, and reusability.

Advantages of High Cohesion:

----------------------------------------

**Better Maintainability** **→** Code is easier to modify and debug.

**Improved Reusability** **→** Well-defined, self-contained modules can be reused in different parts of the application.

**Easier Debugging & Testing** **→** Smaller, focused modules reduce dependencies,making testing simpler.

**Enhanced Readability** **→** Code is more understandable when each module has a clear purpose.

**Reduced Complexity →** Functions and classes handle only relevant tasks, making the system less error-prone.

**→** An interface reference can be used to store objects of any class that implements that interface.

**Downcasting in Java – Compiler & JVM Checks:**

**----------------------------------------------------------------**

When performing downcasting, the compiler and JVM perform certain checks to ensure type safety.

Compiler-Level Checks:

-------------------------------

**Parent-Child Relationship Check:** The reference type and the type being cast must have a parent-child relationship or be of the same type. If they are unrelated types, compiler gives an error: **( Error: incompatible types )** Example:

String s = "Hello";

StringBuffer sb = (StringBuffer) s; // CE: incompatible types

(Here, String and StringBuffer are unrelated classes (they don't share a parent-child relationship), so the compiler throws an error.)

**Reference Type Check:** The reference variable type must be the same as the typecast type or its parent class. If this condition is not met, compiler gives an error: **(** **Error: incompatible types )** Example:

Object o = new String("Hello");

StringBuffer sb = (String) o; // CE: Incompatible types

**JVM-Level Check (Runtime Exception - CCE):** After passing compiler checks, JVM checks the actual runtime object type. If the runtime object is not an instance of the type being cast, Java throws: **ClassCastException (CCE)** Example:

Object o = new String("Hello");

StringBuffer sb = (StringBuffer) o; // Runtime Error: ClassCastException

(Use instanceof to avoid invalid downcasting. Typecasting an object doesn't create a new object, it simply **creates a new reference to the existing object.** In other words, **the object itself remains the same**, but you're just accessing it through a different reference that expects it to be of a certain type.)

Ex: Integer i = new Integer(10);

Number n = (Number) i;

Object o = (Object) n;

System.out.println(i == n); // true

System.out.println(i == o); // true

**Static Flow Control:**

**---------------------------**

Static flow control refers to the sequence of steps the Java Virtual Machine (JVM) follows during the loading, linking, and initialization of a class. It involves the execution of static variables, static blocks, and static methods in a specific order when a class is loaded into memory. Static flow control is determined at compile time and is executed only once per class when the class is first referenced.

**Steps in Static Flow Control:**

**-------------------------------------**

When a class is loaded into memory, the JVM performs the following steps in order:

**Identification of Static Members:** The JVM identifies all static variables, static blocks, and static methods in the class. Static variables are assigned their default values (e.g., 0, null, false).

**Execution of Static Variable Declarations and Static Blocks:** Static blocks and static variable initializations are executed in the order they appear in the code.

**Execution of the main Method (if present):** If the class contains a public static void main(String[] args) method, it is executed after all static variables and static blocks are initialized. Example:

class Test {

// Static variable

static int x = 10;

// Static block 1

static {

System.out.println("Static Block 1: x = " + x);

x = 20;

}

// Static method

static void display() {

System.out.println("Static Method: x = " + x);

}

// Static block 2

static {

System.out.println("Static Block 2: x = " + x);

x = 30;

}

public static void main(String[] args) {

System.out.println("Main Method: x = " + x);

display();

}

}

**Output:**

Static Block 1: x = 10

Static Block 2: x = 20

Main Method: x = 30

Static Method: x = 30

Explanation of the Example:

-------------------------------------

Static Variable Initialization: The static variable x is initialized to its default value (0) and then assigned the value 10.

Static Block 1: The first static block is executed, printing Static Block 1: x = 10 and updating x to 20.

Static Block 2: The second static block is executed, printing Static Block 2: x = 20 and updating x to 30.

Main Method Execution: The main method is executed, printing Main Method: x = 30.

Static Method Execution: The display method is called, printing Static Method: x = 30.

( **Static flow control occurs only once when the class is loaded into memory.** Static variables are assigned default values (0, null, false) before initialization.)

**Write-Only (Indirect Read) State:** When a static variable is declared but not yet assigned a value, Java recognizes it in a write-only (indirect read) state. If we attempt to read this variable inside a static block before its explicit initialization, we get CE: illegal forward reference. Example:

class Test {

static int x = 10; // Explicit initialization

static {

System.out.println(x); // Allowed (direct read after declaration)

System.out.println(y); // CE: illegal forward reference

}

static int y = 20;

}

**Direct Read (Illegal Forward Reference Error):** When a static variable is accessed inside a static block before its declaration, it results in CE: illegal forward reference. This is because the compiler processes static blocks in the order they appear, and the variable hasn’t yet been fully recognized.

**Indirect Read (Allowed in Static Methods):** If we try to access the variable inside a static method (instead of a static block), it is considered an indirect read, and the compiler allows it. Here, the JVM assigns the default value of the variable if it hasn’t been explicitly initialized.

(**Key Points:** Direct read inside a static block before declaration → CE: illegal forward reference.

Indirect read inside a static method → Allowed (returns default value if not initialized).)

**→ Static blocks, also known as static initialization blocks**, are executed only once when the class is loaded into memory by the Java ClassLoader. This happens before any instance of the class is created. Some real world examples:

**1.** When a Java class is loaded, any corresponding native libraries that it depends on must also be loaded. This is typically done using the **System.loadLibrary() method**, which is usually called from a static block. Here's an example:

public class MyClass {

static {

System.loadLibrary("myNativeLibrary");

}

// Native method declaration

public native void myNativeMethod();

}

( In this example, the myNativeLibrary native library is loaded when the MyClass class is loaded. The myNativeMethod()

method is then declared as a native method, which means it is implemented in the native library. By loading the native library in a static block, we ensure that it is loaded only once, when the class is initialized, and that it is available for use by the native methods declared in the class.)

**2.** When you load a database driver class, it typically **includes a static block** that automatically registers the driver with the DriverManager. This means you don't need to explicitly register the driver using the DriverManager.registerDriver() method. Here's an example of how this works:

1. You load the database driver class, e.g., Class.forName("com.mysql.cj.jdbc.Driver").

2. The static block inside the driver class is executed, which registers the driver with the DriverManager.

**→** In Java, you can declare multiple static blocks within a class, and they will be executed in the order they appear, from top to bottom.

**→ System.exit(0) in Java:** The System.exit(int status) method is used to terminate the currently running Java Virtual Machine (JVM). It takes an integer argument (status) that indicates the exit status of the program. By convention:

**0: Indicates a successful termination.**

**Non-zero values: Indicate an abnormal termination (e.g., an error).**

( When System.exit(0) is called, the **JVM terminates immediately**, and no further code is executed. It stops the execution of the program at that point, and control is returned to the operating system.)

**Static Flow Control: Parent to Child:**

**-------------------------------------------------**

When executing a child class the JVM follows this sequence:

**Loading Phase:** Parent class is loaded first (if not already loaded). Child class is loaded next (if not already loaded).

**Initialization Phase:** Parent class is initialized first (static variables and static blocks are executed in the order they appear). Child class is initialized next.

**Execution of main Method:** If the child class has a main method, it is executed. If the child class does not have a main method, the parent class’s main method (if present) is executed.

**( Key Clarifications:** Classes are loaded into memory when first referenced. Loading involves reading the .class file and creating a Class object. Initialization involves executing static variables and static blocks in the order they appear in the code. Parent classes are initialized before child classes. Static initialization is top-down: parent first, then child. If the child class does not have a main method but the parent does, the parent’s main method is executed after initializing both classes.**)**

**→** When you load a child class, the parent class is automatically loaded because the child class inherits members from the parent class. However, when you load a parent class, the child class is not automatically loaded because the parent class does not have a dependency on the child class. The parent class members are available to the child class by default, but the child class members are not available to the parent class by default.

**Instance Control Flow:**

**-------------------------------**

Instance control flow refers to the sequence of steps the Java Virtual Machine (JVM) follows when creating an instance of a class. It involves the execution of instance variables, instance blocks, and constructors in a specific order. Unlike static control flow, which occurs only once when a class is loaded, **instance control flow occurs every time an object of the class is created.**

Key Concepts of Instance Control Flow:

-----------------------------------------------------

**Instance Variables:** Variables declared without the static keyword. They are initialized when an object is created.

**Instance Blocks:** Blocks of code declared without the static keyword. They are executed every time an object is created.

**Constructors:** Special methods used to initialize objects. They are executed after instance variables and instance blocks.

Steps in Instance Control Flow:

-----------------------------------------

When an object of a class is created, the JVM performs the following steps in order:

**Identification of Instance Members:** The JVM identifies all instance variables, instance blocks, and constructors in the class. Instance variables are assigned their default values (e.g., 0, null, false).

**Execution of Instance Variable Declarations and Instance Blocks:** Instance blocks and instance variable initializations are executed in the order they appear in the code.

**Execution of the Constructor:** The constructor is executed to complete the initialization of the object. Example of Instance Control Flow:

class Test {

// Instance variable

int x = 10;

// Instance block 1

{

System.out.println("Instance Block 1: x = " + x);

x = 20;

}

// Constructor

Test() {

System.out.println("Constructor: x = " + x);

x = 30;

}

// Instance block 2

{

System.out.println("Instance Block 2: x = " + x);

x = 40;

}

void display() {

System.out.println("Display Method: x = " + x);

}

}

public class Main {

public static void main(String[] args) {

Test obj = new Test();

obj.display();

}

}

Output:

Instance Block 1: x = 10

Instance Block 2: x = 20

Constructor: x = 40

Display Method: x = 30

Explanation of the Example:

--------------------------------------

Instance Variable Initialization: The instance variable x is initialized to its default value (0) and then assigned the value 10.

Instance Block 1: The first instance block is executed, printing Instance Block 1: x = 10 and updating x to 20.

Instance Block 2: The second instance block is executed, printing Instance Block 2: x = 20 and updating x to 40.

Constructor Execution: The constructor is executed, printing Constructor: x = 40 and updating x to 30.

Display Method Execution: The display method is called, printing Display Method: x = 30.

( Instance variables are assigned default values (0, null, false) before initialization. If the class has a superclass, the superclass constructor is executed before the subclass constructor.)

**→** Object creation can be a costly operation in terms of memory allocation, garbage collection, and performance. If there's no specific requirement to create an object, it's recommended to avoid it. By minimizing unnecessary object creation, you can improve performance, reduce memory usage, and enhance overall system efficiency.

**Steps in Parent-Child Instance Control Flow:**

**------------------------------------------------------------**

**Identification of Instance Members:** The JVM identifies all instance variables and instance blocks in the parent class, then the child class (from top to bottom in the inheritance hierarchy). Parent and child instance variables are assigned default values (e.g., 0, null, false).

**Parent Class Initialization:** Parent instance variables are initialized and instance blocks are executed in the order they appear in the code. **→ Parent constructor is executed.**

**Child Class Initialization:** Child instance variables are initialized and instance blocks are executed in the order they appear in the code. **→** **Child constructor is executed.**

( **Key Points:** The child constructor implicitly calls super() (parent constructor) as its first statement. Instance variables are initialized to default values (0, null, false) before explicit initialization.)

**→ From a static context**, such as a static block or a static method, you cannot access instance members (instance variables or instance methods) directly. This is because instance members belong to a specific instance of the class, whereas static contexts belong to the class itself, not to any particular instance. When the JVM executes a static block or method, it doesn't have an instance of the class to reference, so it can't resolve instance members.

**→ There are five standard ways** to create an object in Java. Here’s a brief explanation of each:

**Using new Operator (Most Common Way):**

Test obj = new Test(); // Object created using new operator

**Using Class.forName().newInstance() (Reflection API, Deprecated in Java 9) :**

Test obj = (Test) Class.forName("Test").newInstance(); // Object created using reflection

( This approach is now deprecated in Java 9+; use **Class.getDeclaredConstructor().newInstance()** instead.)

**Using Factory Method (Common in Singleton Pattern):**

class Test {

private static Test instance = new Test(); // Eager initialization

private Test() {} // Private constructor

public static Test getInstance() { // Factory method

return instance;

}

}

public class Main {

public static void main(String[] args) {

Test obj = Test.getInstance(); // Object created using a factory method

}

}

**Using clone() Method (Cloning an Existing Object, Requires Cloneable Interface):**

Test obj1 = new Test();

Test obj2 = (Test) obj1.clone(); // Object created using clone()

**Using Deserialization (Restoring an Object from a Serialized Stream):**

// Deserialize the object

ObjectInputStream in = new ObjectInputStream(new FileInputStream("obj.ser"));

Test obj2 = (Test) in.readObject(); // Object created using deserialization

in.close();

**( Key Takeaways:** new is the most common way to create objects. Reflection (newInstance()) is powerful but deprecated in

modern Java. Factory methods are used in design patterns like Singleton. clone() is used to create an exact copy, but requires Cloneable. Deserialization is useful for restoring objects from files/networks**.)**

**Constructor in Java:**

**---------------------------**

A constructor is a special method in Java that is **used to initialize objects**. It is called automatically when an object of a class is created. Constructors have the same name as the class and do not have a return type, not even void.

Constructors are primarily used to initialize the state of an object (e.g., setting initial values for instance variables). A class can have multiple constructors with different parameter lists (constructor overloading).

Types of Constructors:

------------------------------

**Default Constructor:** If no constructor is defined in a class, the Java compiler automatically provides a default constructor.

**Parameterized Constructor:** A constructor with parameters is called a parameterized constructor. It is used to initialize instance variables with specific values. Example:

class Student {

String name;

int age;

// Parameterized constructor

Student(String name, int age) {

this.name = name;

this.age = age;

}

}

**Copy Constructor:** A constructor that takes an object of the same class as a parameter and copies its values is called a copy constructor.

Key Points About Constructors: A constructor can call another constructor in the same class using this() or in the parent class using super().

**this():** Used to call another constructor in the same class.

**super():** Used to call a constructor in the parent class. **Example:**

class Parent {

Parent() {

System.out.println("Parent constructor called.");

}

}

class Child extends Parent {

Child() {

super(); // Calls Parent constructor

System.out.println("Child constructor called.");

}

}

**Real-World Use Case:** Constructors are commonly used to:

------------------------------------------------------------------------------

1. Initialize object state (e.g., setting default values for instance variables).
2. Perform setup tasks (e.g., opening database connections).
3. Enforce constraints (e.g., validating input parameters).

**→** Instance blocks (also known as instance initializer blocks) are used when you want to execute some logic during object creation other than just initialization. They are useful if you want to run common logic for multiple constructors. Example of using an instance block to increment a count in a database every time an object is created.

**→** While both constructors and instance initializer blocks can be used to initialize instance variables, they serve distinct

purposes, and they are not always interchangeable. Replacing one with the other might work in some simple cases, but it won't be a suitable solution in all situations.

**→** If we mistakenly declare a return type for a constructor, the **compiler treats it as a method, not a constructor.** While it's technically possible to have a method in a class with the same name as the class itself, it's strongly discouraged and considered very bad practice. It can lead to significant confusion and ambiguity. Example:

class Test {

// Mistakenly adding a return type

void Test() {

System.out.println("This is a method, not a constructor");

}

}

**→** In Java, the only applicable modifiers for a constructor are: **public, private, default (no modifier), protected.** If you try to use any other modifier, such as static, final, abstract, or synchronized, you will get a compile-time error (CE).

**→ Compiler, not JVM, generates the default constructor** if no constructor is explicitly defined in the class. If at least one constructor is written, the compiler does not generate a default constructor.

Every class in Java must have at least one constructor, the default constructor (generated by the compiler if no constructor is written). A constructor written by the programmer. A class cannot have both the compiler-generated default constructor and a programmer-defined constructor at the same time.

**Characteristics of the Default Constructor:** Always a no-argument constructor. Access modifier is the same as the class (either public or default/package-private). Contains only one statement: super();

This calls the no-arg constructor of the superclass (if available). Generated by the compiler only when no constructor is explicitly defined.

**→ The very first statement** in a constructor must be a call to **either this()** (another constructor in the same class) **or super()** (the constructor of the superclass). If you don't explicitly write either this() or super(), the compiler **automatically inserts a call to super()** as the first statement. Within a constructor, you can call either super() or this(), **but not both simultaneously.**

**→ super() and this() (when used as constructor calls) are exclusively allowed within constructors.** Attempting to use them anywhere else in the code (like inside a regular method, an instance initializer block, or a static initializer block) will result in a compile-time error.

**Difference Between this(), super() and this, super:**

**---------------------------------------------------------------------**

**this() and super():** Purpose: **Used for constructor chaining.** Context: Must be the first statement and can only be used once in a constructor.

**this and super:** Purpose: **These are keywords** that can be used to access members of the current object or the superclass, respectively. this and super can be used anywhere in an instance method or constructor (not restricted to the first statement), **except in static areas.** There's no limit to the number of times you can use this and super in a method or constructor.

**→ Constructor overloading is possible in Java,** as multiple constructors can be declared within a class with different parameter lists. This follows the method overloading concept, where the compiler differentiates constructors based on the number and type of parameters. Overloaded constructors provide flexibility in object creation by providing multiple ways to initialize an object. Example:

class Person {

String name;

int age;

// Constructor with one argument

Person(String name) {

this.name = name;

}

// Constructor with two arguments

Person(String name, int age) {

this.name = name;

this.age = age;

}

}

**→** When you call a constructor with an argument of a certain type, Java will try to find a constructor that matches the argument type exactly. If no exact match is found, Java will attempt to promote the argument type to a compatible type, such as: **byte → short → int → long → float → double, char → int** If a constructor with a promoted type is found, Java will call that constructor automatically.

**→ Constructors are not inherited** in the same way that methods and variables are. When you create an object of a subclass, the subclass's constructor is responsible for initializing the object, including the inherited members from the superclass. While the superclass constructor is called (either explicitly or implicitly using super()), the constructor itself is not inherited. **Each class has its own constructors.** Since constructors are not inherited, the concept of overriding also doesn't apply to them. Overloading is applicable to constructors.

**→** Every class in Java, including abstract classes, has at least one constructor. If you don't explicitly define a constructor in your class, the compiler automatically provides a default no-argument constructor. **Interfaces do not have constructors.**

**→** Recursive method calls lead to a runtime exception (StackOverflowError), whereas recursive constructor calls result in a compile-time error as Java prevents infinite constructor invocation.

**→** If a parent class contains a parameterized constructor, then while defining a child class, we must take special care regarding constructors:Compiler does not generate a default constructor if any constructor is explicitly defined. If the parent class has only a parameterized constructor, the child class must **explicitly call super(arguments)**, or it will result in a compile-time error. It is recommended to define a no-argument constructor to avoid such issues.

**→ If a parent class constructor throws a checked exception**, then the child class constructor must either: Declare the **same exception** in its throws clause **or a parent** exception of the thrown exception.

**Singleton Class in Java:** A singleton class is a class that allows only one instance to be created and provides a global access point to that instance. Requirement of Singleton Class:

**Resource Management** **→** Ensures controlled access to shared resources like database connections, loggers, or configuration managers.

**Global Access** **→** Provides a single access point to a specific object across the application.

**Prevention of Multiple Instances** **→** Helps avoid unnecessary memory consumption by restricting multiple object creations.

**Advantages of Singleton Class:**

**------------------------------------------**

Saves Memory **→** Only one object is created, reducing memory overhead.

Prevents Conflicts **→** Ensures consistency by maintaining a single instance.

Thread Safety **→** Can be implemented in a thread-safe way to avoid concurrency issues.

Global Access Point **→** Makes it easier to access common resources.

Here are two common approaches to implement a Singleton:

**1. Eager Initialization (Early Object Creation):** The instance is created when the class is loaded, even if it’s not used immediately. This is thread-safe by default. Example: The Java Runtime class

public class Runtime {

private static final Runtime currentRuntime = new Runtime();

public static Runtime getRuntime() {

return currentRuntime;

}

private Runtime() {}

}

// The Java Runtime class is a classic example of eager initialization.

**Pros:** Simple and thread-safe.

**Cons:** The instance is created even if it’s never used, which may waste resources.

**Use Case:** When the instance is always needed.

**2. Lazy Initialization (Object Created When Needed):** The instance is created only when it’s first requested. To make this thread-safe, use double-checked locking. Example:

public class DatabaseConnection {

// Use volatile to ensure visibility across threads

private static volatile DatabaseConnection instance;

// Private constructor

private DatabaseConnection() {

// Initialization code (e.g., connect to the database)

}

// Global access point with double-checked locking

public static DatabaseConnection getInstance() {

if (instance == null) { // First check (no locking)

synchronized (DatabaseConnection.class) { // Lock the class

if (instance == null) { // Second check (inside the lock)

instance = new DatabaseConnection();

}

}

}

return instance;

}

}

**Pros:** Saves resources by creating the instance only when needed.

**Cons:** Requires synchronization for thread safety.

**Use Case:** When the instance is resource-heavy or rarely used.

**→ By declaring every constructor as private,** we can restrict child class creation without using final. If all constructors are private, no other class can extend it, because the child class must call super(), which it cannot access. This is commonly used in Singleton Design Pattern.

class Parent {

private Parent() { // Private constructor

System.out.println("Parent constructor called");

}

}

// This will cause a CE: Constructor 'Parent()' is not accessible

class Child extends Parent {} // Child class cannot be created, preventing inheritance.

**→** Java supports the following types of inheritance:

**Single Inheritance:** A class inherits from only one superclass. This is the most common and straightforward type of inheritance.

**Multilevel Inheritance:** A class inherits from a superclass, which in turn inherits from another superclass, forming a hierarchy of inheritance.

**Hierarchical Inheritance:** Multiple classes inherit from the same superclass.

( Java does not support **multiple inheritance through classes.** This means a class cannot directly inherit from more than one superclass.)

**→** To compile a Java program for a specific version of Java, you can use the **-source** option with the javac command. The syntax is as follows: javac -source <version> <filename>.java

For example, to compile a Java program for Java 8: javac -source 1.8 MyProgram.java

This tells the compiler to generate bytecode that is compatible with Java 8. Note that the -source option only affects the language features and syntax that are allowed in the code. It does not affect the runtime environment or the libraries that are used. Also, you can use the **-target** option to specify the target JVM version. For example: javac -source 1.8 -target 1.8 MyProgram.java

This tells the compiler to generate bytecode that is compatible with Java 8, and to use the Java 8 runtime environment.

**Exception Handling**

**====================================================================================**

**An exception is an unexpected event** that occurs during the execution of a program, disrupting its normal flow. Exceptions

are typically caused by errors in the program logic, invalid input, or external factors (e.g., file not found, network issues).

In Java, exceptions are represented as objects of classes that extend the Throwable class. The two main types of exceptions are: **Checked Exceptions** (e.g., IOException, SQLException) and **Unchecked Exceptions** (e.g., NullPointerException, ArithmeticException).

Exception Handling in Java:

-------------------------------------

**Exception handling is a mechanism to handle runtime errors gracefully**, ensuring that the program does not crash abruptly. It involves:

Detecting Exceptions: Identifying when an exception occurs.

Handling Exceptions: Taking appropriate actions (e.g., logging, recovery, or termination).

**→** Java provides five keywords for exception handling: **try, catch, finally, throw, throws**

Exception Handling Workflow:

------------------------------------------

When an exception occurs in the try block, the JVM creates an exception object. The JVM looks for a matching catch block to handle the exception. If a matching catch block is found, it executes the code inside it. The finally block (if present) is executed, regardless of whether an exception occurred. If no matching catch block is found, the program terminates abruptly. Example:

public class Main {

public static void main(String[] args) {

try {

int result = 10 / 0; // ArithmeticException

System.out.println("Result: " + result);

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e.getMessage());

} finally {

System.out.println("Finally block executed.");

}

}

}

Output: Exception caught: / by zero

Finally block executed.

Key Concepts of Exception Handling:

-------------------------------------------------

**try Block:** Contains code that might throw an exception. **Must be followed by at least one catch or finally block.**

**catch Block:** Handles the exception thrown by the try block. Can have multiple catch blocks to handle different types of exceptions.

**finally Block:** Executes code regardless of whether an exception occurs. Used for cleanup tasks (e.g., closing files, releasing resources).

**throw Keyword:** Used to explicitly throw an exception.

**throws Keyword:** Declares exceptions that a method might throw.

Example: throw and throws

-------------------------------------

public class Main {

static void validateAge(int age) throws ArithmeticException {

if (age < 18) {

throw new ArithmeticException("Age must be 18 or above.");

} else {

System.out.println("Valid age.");

}

}

public static void main(String[] args) {

try {

validateAge(15); // Throws ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e.getMessage());

}

}

}

Output: Exception caught: Age must be 18 or above.

Real-World Use Cases:

------------------------------

**File Handling:** Handle IOException when reading/writing files.

**Database Operations:** Handle SQLException when interacting with databases.

**Input Validation:** Handle NumberFormatException when parsing user input.

Advantages of Exception Handling:

----------------------------------------------

**Graceful Termination:** Prevents the program from crashing abruptly.

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

**Debugging:** Provides detailed information about errors for debugging.

**→ For every thread, JVM creates a runtime stack.** Each method call performed by the thread is stored in the corresponding stack, and each entry in the stack is called a **stack frame or activation record.** After completing a method call, the corresponding entry is removed from the stack. Once all methods have executed, the stack becomes empty, and the JVM destroys it just before terminating the thread.

**→ Inside a method, if an exception occurs, the method where it was raised is responsible for creating an exception object containing: Exception name, Description, Stack trace** (location of the exception occurrence)

After creating the exception object, the method hands it over to the JVM. The JVM then checks whether the method has any exception handling code. If handling code is present, the exception is caught and managed accordingly.

If not, the method terminates abnormally, and its corresponding stack frame is removed. The JVM then checks the caller method for handling code. If the caller method also lacks exception handling, it too terminates abnormally, and its stack frame is removed. This process continues up the call stack until the main method.

If the main method does not handle the exception, it is also terminated abnormally, and its stack frame is removed. At this point, the JVM hands over exception handling to the default exception handler, which is part of the JVM. The default exception handler prints the exception details in the following format and terminates the program abnormally:

Exception in thread "main" java.lang.ExceptionName: description

at ClassName.methodName(FileName.java:lineNumber)

at ClassName.methodName(FileName.java:lineNumber)

…

**→** If a method throws an exception, but it is caught and handled within the program (including in the main method), and the main method subsequently completes normally, the program's termination is still considered normal. If an exception propagates all the way up to the main method and is not caught, the program terminates abnormally. The exception's stack trace is usually printed to the console. The main method's completion (without uncaught exceptions) or the explicit System.exit() status code as the primary indicators of normal or abnormal termination.

**Exception Hierarchy in Java:** In Java, the exception hierarchy is **rooted in the Throwable class.** All exceptions and errors are

subclasses of Throwable. The hierarchy is divided into two main branches:

**Exception:** Represents conditions that a program might want to catch and handle.

**Error:** Represents serious issues that are typically beyond the control of the program.

Here’s the detailed hierarchy:

----------------------------------------

**Throwable Class:** The root class for all exceptions and errors in Java. Contains methods like getMessage(), printStackTrace(), and toString() for error handling.

**Exception Class:** Represents conditions that a program might want to catch and handle. Divided into two categories:

**Checked Exceptions:** Must be handled at compile time (e.g., IOException, SQLException).

**Unchecked Exceptions:** Occur at runtime and do not need to be explicitly handled (e.g., NullPointerException, ArithmeticException).

Subclasses of Exception:

---------------------------------

**IOException:** For input/output errors (e.g., file not found).

**SQLException:** For database-related errors.

**RuntimeException:** The superclass for all unchecked exceptions.

**NullPointerException:** Occurs when trying to access a null object.

**ArithmeticException:** Occurs during invalid arithmetic operations (e.g., division by zero)

**ArrayIndexOutOfBoundsException:** Occurs when accessing an invalid array index.

**Error Class:** Represents serious issues that are typically beyond the control of the program. Errors are not meant to be caught or handled by the program. Subclasses of Error:

**OutOfMemoryError:** Occurs when the JVM runs out of memory.

**StackOverflowError:** Occurs when the stack overflows (e.g., infinite recursion).

**VirtualMachineError:**  Indicates issues with the JVM (e.g., resource exhaustion).

**Exception Hierarchy Diagram:**

**Throwable**

── **Exception**

**│** **──** IOException

│ ── SQLException

│ ── **RuntimeException**

│ ── NullPointerException

│ ── ArithmeticException

│ ── ArrayIndexOutOfBoundsException

└── **Error**

── OutOfMemoryError

── StackOverflowError

── VirtualMachineError

**Key Differences Between Exception and Error:**

**Aspect Exception Error**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Type Represents recoverable conditions. Represents unrecoverable conditions.

Handling Can be caught and handled by the program. Should not be caught or handled.

Examples IOException, NullPointerException. OutOfMemoryError, StackOverflowError.

**Real-World Use Cases:**

**-------------------------------**

**Checked Exceptions:** Handle file I/O errors (IOException). Handle database errors (SQLException).

**Unchecked Exceptions:** Handle invalid user input (NumberFormatException).

Handle null object access (NullPointerException).

**Errors:** Log and terminate the program in case of OutOfMemoryError or StackOverflowError.

**Difference Between Checked and Unchecked Exceptions in Java:**

**---------------------------------------------------------------------------------------**

**Checked Exceptions:** Checked exceptions are exceptions that are checked **at compile time**. The compiler ensures that these exceptions are either: **Handled using a try-catch block or Declared using the throws keyword in the method signature.**

**Examples:** IOException (e.g., file not found). SQLException (e.g., database connection failure). ClassNotFoundException (e.g., class not found).

**→** Used for conditions that a program can recover from (e.g., file not found, network issues). Must be handled explicitly, or the code will not compile. Example of Checked Exception:

import java.io.FileInputStream;

import java.io.FileNotFoundException;

public class Main {

public static void main(String[] args) {

try {

FileInputStream file = new FileInputStream("nonexistent.txt");

} catch (FileNotFoundException e) {

System.out.println("Checked Exception: " + e.getMessage());

}

}

}

Output: Checked Exception: nonexistent.txt (No such file or directory)

**Unchecked Exceptions:** Unchecked exceptions are exceptions that are not checked at compile time. They occur at runtime and are subclasses of RuntimeException. Examples: NullPointerException (e.g., accessing a null object), ArithmeticException (e.g., division by zero).

Used for conditions that are typically programming errors (e.g., null pointer, invalid arithmetic). Handling is optional. If not handled, the program will terminate abruptly. Example:

public class Main {

public static void main(String[] args) {

try {

int result = 10 / 0; // ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Unchecked Exception: " + e.getMessage());

}

}

}

Output: Unchecked Exception: / by zero

Key Differences:

**Aspect Checked Exceptions Unchecked Exceptions**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Checked at Compile time. Runtime.

Handling Must be handled explicitly (try-catch or throws). Handling is optional.

Recovery Represents recoverable conditions. Represents programming errors.

Superclass Subclasses of Exception Subclasses of RuntimeException and Error.

(excluding RuntimeException and error)

**→ Both checked and unchecked exceptions occur during runtime, not during compile-time.**

**→** **RuntimeException** (and all its subclasses) and **Error** (and all its subclasses) are considered unchecked exceptions. All

other exceptions that are not subclasses of RuntimeException or Error are considered checked exceptions.

**→ A checked exception is fully checked** if all its child classes are also checked. Examples: IOException, InterruptedException (since all their subclasses are checked).

**A checked exception is partially checked** if it has at least one unchecked child. The only two partially checked exceptions in Java are: **Throwable** (parent of both Exception and Error) and **Exception** (parent of both checked exceptions and RuntimeException, which is unchecked)

**→ It's strongly recommended** (and often required for checked exceptions) to handle exceptions appropriately. Code that might throw an exception is often referred to as "risky code" or "exception-prone code." This code should be placed within a try block. The corresponding code that handles the exception (if it occurs) should be placed in one or more catch blocks.

**→** If an exception occurs inside a try block, the remaining statements in that block won't execute, even if the exception is handled. To improve readability and maintainability, the try block should contain only risky code, keeping its length as short as possible. **Exceptions can also occur inside catch and finally blocks.** If an exception arises in any statement outside the try block, it always leads to abnormal termination, unless handled separately.

In Java, exceptions can be printed using **three main methods:**

**printStackTrace():** Prints the name of the exception, description, and complete stack trace (including method calls leading to the exception). This is the most detailed way to print exceptions. Use printStackTrace() when debugging, as it provides a detailed stack trace. Example:

try {

int x = 10 / 0;

} catch (ArithmeticException e) {

e.printStackTrace();

}

Output: java.lang.ArithmeticException: / by zero

at Main.main(Main.java:3)

**toString():** Prints the exception class name and exception message (without stack trace). Use toString() when you need a brief description of the exception. Example:

try {

int x = 10 / 0;

} catch (ArithmeticException e) {

System.out.println(e.toString());

}

Output: java.lang.ArithmeticException: / by zero

**getMessage():** Prints only the exception message (without class name or stack trace). Use getMessage() when you only need the specific error message. Example:

try {

int x = 10 / 0;

} catch (ArithmeticException e) {

System.out.println(e.getMessage());

}

Output: / by zero

**→** toString() and getMessage() require explicit printing. printStackTrace(), on the other hand, prints the stack trace

implicitly, without needing an explicit System.out.println() statement. You can simply call exception.printStackTrace().

**→** The default exception handler's core functionality is to call the **printStackTrace() method of the exception object.**

**→** Handling exceptions varies based on the type of exception, and it's highly recommended to use separate catch blocks for different exception types.

**Try with Multiple Catch Blocks:** We can write multiple catch blocks for a single try block to handle different exceptions separately. This is recommended because different exceptions may require different handling logic. Example:

public class MultipleCatchExample {

public static void main(String[] args) {

try {

int[] arr = new int[5];

arr[10] = 100; // ArrayIndexOutOfBoundsException

int x = 10 / 0; // ArithmeticException

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index is out of bounds: " + e.getMessage());

} catch (ArithmeticException e) {

System.out.println("Cannot divide by zero: " + e.getMessage());

} catch (Exception e) {

System.out.println("Some other exception occurred: " + e.getMessage());

}

}

}

Output: Array index is out of bounds: Index 10 out of bounds for length 5

**(NOTE:** Since an exception occurs in the first line inside try, the second line (10/0) is never executed.**)**

**Key Points: Order Matters:** More specific exceptions should be caught before general ones like Exception, otherwise, it will cause a **compile-time error**. Example of incorrect order:

try {

int x = 10 / 0;

} catch (Exception e) { // This should be last

System.out.println("General Exception");

} catch (ArithmeticException e) { // Unreachable code (CE)

System.out.println("Cannot divide by zero");

}

From Java 7 Onwards: **Use Multi-Catch Block (| Operator):** If multiple exceptions have the same handling logic, we can combine them using | (multi-catch). Example:

try {

int[] arr = new int[5];

arr[10] = 100; // ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException | ArithmeticException e) {

System.out.println("Exception occurred: " + e.getMessage());

}

**Recommended Approach:** Always separate catch blocks for different exceptions when different handling is needed. Use multi-catch (|) when handling logic is the same.

**→** You cannot declare two catch blocks for the same exception type. If you try to do so, you will get a compile-time error.

**Difference Between final, finally, and finalize in Java:**

**------------------------------------------------------------------------**

These three keywords in Java (final, finally, and finalize) serve completely different purposes. Here's a detailed explanation of each:

**final:**

**-------**

**Purpose:** Used to define immutability or restrict modification.

**Applicable To:** Variables, methods, and classes.

Context Description:

----------------------------

final variable: The value of the variable cannot be changed (constant).

final method: The method cannot be overridden in a subclass.

final class: The class cannot be extended (cannot have subclasses). **Example:**

final int x = 10; // final variable

x = 20; // Error: Cannot assign a value to final variable 'x'

final class Parent {} // final class

class Child extends Parent {} // Error: Cannot inherit from final class 'Parent'

**finally:**

**----------**

**Purpose:** Used in exception handling to define a block of code that always executes, regardless of whether an exception occurs. Even if a return statement is present in the try or catch block, the finally block still executes. **The finally block has higher priority than the return statement.**

**Applicable To:** try-catch blocks. Example:

public class Main {

public static void main(String[] args) {

try {

int result = 10 / 0; // ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Exception caught: " + e.getMessage());

} finally {

System.out.println("Finally block executed.");

}

}

}

Output: Exception caught: / by zero

Finally block executed.

**finalize:**

**-----------**

**Purpose:** A method in the Object class that is called by the garbage collector before an object is destroyed. It is used for cleanup activities (e.g., releasing resources).

Note: The finalize method is deprecated in **Java 9** due to performance issues and unpredictable behavior and should not be relied upon for resource cleanup. Use try-with-resources or explicit cleanup methods instead. Example:

public class MyClass {

@Override

protected void finalize() throws Throwable {

System.out.println("Finalize method called.");

}

public static void main(String[] args) {

MyClass obj = new MyClass();

obj = null; // Make the object eligible for garbage collection

System.gc(); // Request garbage collection

}

}

Output (may vary): Finalize method called.

**Summary:**

**--------------**

**final:** Used for immutability and restricting modification.

**finally:** Ensures code execution in exception handling.

**finalize:** Deprecated method for cleanup before object destruction.

**→** finally cleans up resources used in try-catch and finalize cleans up resources associated with an object.

**→ Prefer try-with-resources** (from Java 7+) instead of manually closing resources in finally.

try (FileInputStream fis = new FileInputStream("data.txt")) {

// File operations

} catch (IOException e) {

e.printStackTrace();

}

(This ensures the file is closed automatically!)

**Rules for try-catch-finally in Java:**

**---------------------------------------------**

1. **A try block must always be followed by either a catch block or a finally block.** Example:

try { } catch (Exception e) { }

try { } finally { }

try { } catch (Exception e) { } finally { }

Invalid (Compile-time error - "Try without catch or finally"):

try { } // CE: 'try' without 'catch' or 'finally'

1. **A catch or finally block cannot exist without a preceding try block.**

Invalid (Compile-time error - "Catch without Try"): catch (Exception e) { } // CE

finally { } // CE

1. **Curly braces {} are mandatory for try, catch, and finally blocks.** Invalid (Compile-time error):

try

System.out.println("Hello"); // CE: Try without {}

catch (Exception e)

System.out.println("Handled"); // CE: Catch without {}

1. **try-catch-finally blocks can be nested (Nested Exception Handling is allowed).**
2. **finally must always be the last block:** If you place finally before catch, the compiler will treat catch as an independent block, leading to a compile-time error because catch without try is invalid. Example:

try {

int x = 10 / 0; // Exception occurs

} finally {

System.out.println("Finally block executed");

} catch (ArithmeticException e) { // CE: 'catch' without 'try'

System.out.println("Catch block executed");

}

1. After a try block, you can have multiple catch blocks to handle different types of exceptions. **However, there can be at most one finally block.** If you define multiple finally blocks, the compiler will treat each additional finally block as a separate finally block without a corresponding try block. This will result in a compile-time error (CE).

**→** The **throw keyword** is used to explicitly create and hand over an exception object to the JVM.

**Key Points About throw:**

**----------------------------------**

It is used inside a method or block to throw an exception manually. The thrown exception must be an object of Throwable or its subclasses (Exception or Error). Execution stops immediately after throw, and the JVM looks for a matching catch block. Checked exceptions must be either caught or declared using throws. Example:

class Test {

static void validate(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be 18 or above");

}

System.out.println("Access granted");

}

public static void main(String[] args) {

validate(15); // Exception is thrown manually

}

}

Output: Exception in thread "main" java.lang.IllegalArgumentException: Age must be 18 or above

**→** The best use of the throw keyword is for user-defined (customized) exceptions, where a programmer explicitly creates

an exception object and hands it over to the JVM.

Steps to Create a Custom Exception Using throw:

-----------------------------------------------------------------

Create a custom exception class by extending Exception (for checked exceptions) or RuntimeException (for unchecked exceptions). Provide a constructor to initialize the exception message. Use the throw keyword to explicitly throw the custom exception. Example:

// Step 1: Create a user-defined exception

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

// Step 2: Use throw to create and throw the exception

class Test {

static void validate(int age) throws InvalidAgeException {

if (age < 18) {

throw new InvalidAgeException("Age must be 18 or above to vote");

}

System.out.println("Eligible to vote");

}

public static void main(String[] args) {

try {

validate(15); // Throwing custom exception

} catch (InvalidAgeException e) {

System.out.println("Exception Caught: " + e.getMessage());

}

}

}

Output: Exception Caught: Age must be 18 or above to vote

Why Use throw for Custom Exceptions?

-----------------------------------------------------

**Better Code Readability** – Makes exception handling clear and meaningful.

**Encapsulation of Business Logic** – Helps enforce application-specific rules.

**Custom Messages** – Provides meaningful error messages to users.

**Controlled Exception Flow** – Allows handling errors in a structured way.

**Conclusion:** Using throw for user-defined exceptions improves error handling and makes debugging easier.

**→** If we declare an exception reference variable but do not initialize it before throwing, it will result in a NullPointerException at runtime. This happens because the reference variable is null, and attempting to throw null causes the JVM to fail when handling the exception. Therefore, it is essential to initialize an exception object before using the throw keyword. Example:

class Test {

static ArithmeticException e; // Reference declared but not initialized

public static void main(String[] args) {

throw e; // NullPointerException

}

}

**→** After a throw statement, writing any statement directly will result in a compile-time error (CE: unreachable statement) because the throw statement causes immediate transfer of control, making any subsequent code unreachable.

**→ The throw keyword can only be used with objects of Throwable types** (i.e., subclasses of Throwable). If we try to use it with a normal Java object, the compiler will throw a compile-time error (CE: incompatible types).

**throws Keyword in Java:** The throws keyword is used in method signatures to declare exceptions that the method might

throw. It indicates that the method does not handle the exception itself but instead passes the responsibility of handling the exception to the caller of the method.

Key Features of throws:

--------------------------------

**Exception Declaration:** The throws keyword is used to declare **checked exceptions** that a method might throw.

**Caller Responsibility:** The caller of the method must handle the declared exception using a try-catch block or propagate it further using throws.

**Multiple Exceptions:**  A method can declare multiple exceptions separated by commas.

**Unchecked Exceptions:** Unchecked exceptions (e.g., NullPointerException, ArithmeticException) do not need to be declared

using throws.

**Syntax:** returnType methodName(parameters) throws Exception1, Exception2, ... {

// Method body

}

( It is commonly used in file handling, database operations, and network programming.)

**→** However, using the throws keyword only informs the compiler about the potential exception, but it does not prevent the abnormal termination of the program. When a method or constructor throws an exception, the program will still terminate abruptly, unless the exception is caught and handled by a try-catch block.

**→** If a method at the deepest level of a multilevel call stack throws a checked exception but does not handle it using try-catch or declare it with throws, the compiler will generate a compile-time error (CE: unreported exception; must be caught or declared to be thrown). This happens even if the main method handles the exception because each intermediate method must either handle the exception or propagate it using throws.

**→** While there are cases where throws is more suitable, it's generally recommended to use try-catch blocks for handling exceptions, as they provide better error handling, code organization, and flexibility.

**→ The throws keyword in Java is used only for methods and constructors. It is not allowed for classes or interfaces.** Why not for Classes/Interfaces: Classes and interfaces themselves don't execute code that can directly throw checked exceptions. They are declarations or blueprints. The methods and constructors within classes are the ones that might throw exceptions. Therefore, it's the methods and constructors that use the throws keyword, not the classes or interfaces themselves.

**→** The throws keyword, when used in a method or constructor declaration, must be followed by a type that is a subclass of Throwable. This means it must be either Exception (or a subclass of Exception) or Error (or a subclass of Error). If you try to use the throws keyword with a type that is not a Throwable (i.e., a regular Java class that doesn't inherit from Throwable), you will get a compile-time error.

Case 1:

public static void main(String[] args) {

throw new Exception();

}

Case 2:

public static void main(String[] args) {

throw new Error();

}

In the first case, throw new Exception(); results in a compile-time error (CE: unreported exception; must be caught or declared to be thrown) because Exception is a checked exception, and Java requires checked exceptions to be either caught or declared using throws.

In the second case, throw new Error(); leads to a runtime exception (RE) because Error is an unchecked exception (a subclass of Throwable that is not Exception), and Java does not enforce compile-time checks for unchecked exceptions.

**→** If there is no possibility of an exception occurring within a try block, we cannot write a catch block for that exception; otherwise, we get a compile-time error (CE: Exception XXX is never thrown in body of corresponding try statement).

**However, this rule applies only to fully checked exceptions.** For unchecked exceptions (like RuntimeException) and partially checked exceptions (like Throwable or Exception), the compiler does not enforce this rule.

Invalid Case (Checked Exception is not thrown):

---------------------------------------------------------------

import java.io.\*;

public class Test {

public static void main(String[] args) {

try {

System.out.println("No exception thrown");

} catch (IOException e) { // Compilation Error

System.out.println("Handled IOException");

}

}

}

**Error:**

Test.java:7: error: exception IOException is never thrown in body of corresponding try statement

} catch (IOException e) {

^

Unchecked Exception (Allowed):

-------------------------------------------

public class Test {

public static void main(String[] args) {

try {

System.out.println("No exception thrown");

} catch (ArithmeticException e) { // No error (unchecked exception)

System.out.println("Handled ArithmeticException");

}

}

}

// Since ArithmeticException is an unchecked exception, the compiler does not enforce this rule, and the code compiles fine.

**Customized or User-Defined Exceptions in Java:**

**-----------------------------------------------------------------**

A customized or user-defined exception is an exception that you create to handle specific errors or conditions in your application. Java allows you to define your own exception classes by extending the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions).

Why Use Custom Exceptions?

----------------------------------------

**Specific Error Handling:** Custom exceptions allow you to handle specific errors in a meaningful way.

**Improved Readability:** They make your code more readable and maintainable by clearly indicating the type of error.

**Reusability:** You can reuse custom exceptions across your application.

**Better Debugging:** Custom exceptions provide detailed error messages and context for debugging.

Steps to Create a Custom Exception:

------------------------------------------------

**Create a Class:** Extend the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions).

**Define Constructors:** Provide constructors to initialize the exception with a custom error message.

**Throw the Exception:** Use the throw keyword to throw the custom exception in your code.

Example of a Custom Checked Exception:

// Custom checked exception

class InsufficientFundsException extends Exception {

// Constructor with a custom message

InsufficientFundsException(String message) {

super(message);

}

}

class BankAccount {

private double balance;

BankAccount(double balance) {

this.balance = balance;

}

void withdraw(double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Insufficient funds. Balance: " + balance);

} else {

balance -= amount;

System.out.println("Withdrawal successful. Remaining balance: " + balance);

}

}

}

public class Main {

public static void main(String[] args) {

BankAccount account = new BankAccount(1000);

try {

account.withdraw(1500); // Throws InsufficientFundsException

} catch (InsufficientFundsException e) {

System.out.println("Error: " + e.getMessage());

}

}

}

Output: Error: Insufficient funds. Balance: 1000.0

Example of a Custom Unchecked Exception:

// Custom unchecked exception

class InvalidAgeException extends RuntimeException {

// Constructor with a custom message

InvalidAgeException(String message) {

super(message);

}

}

class Voter {

void validateAge(int age) {

if (age < 18) {

throw new InvalidAgeException("Invalid age. Must be 18 or above.");

} else {

System.out.println("Valid age. You are eligible to vote.");

}

}

}

public class Main {

public static void main(String[] args) {

Voter voter = new Voter();

voter.validateAge(15); // Throws InvalidAgeException

}

}

Output: Exception in thread "main" InvalidAgeException: Invalid age. Must be 18 or above.

Key Points About Custom Exceptions:

--------------------------------------------------

**Checked vs Unchecked:** Extend Exception for checked exceptions (must be handled or declared). Extend RuntimeException for unchecked exceptions (optional handling).

**Constructors:** Provide at least a constructor that accepts a String message.

**Meaningful Messages:** Use descriptive error messages to make debugging easier.

**Reusability:** Define custom exceptions in a separate package for reusability across your application.

**Real-World Use Cases:**

**-------------------------------**

Banking Applications: Custom exceptions like InsufficientFundsException or InvalidAccountException.

E-Commerce Applications: Custom exceptions like ProductNotFoundException or InvalidPaymentException.

Validation: Custom exceptions like InvalidInputException or ValidationException.

**(Custom exceptions allow you to handle specific errors in a meaningful way.)**

**→** The throw keyword is best suited for user-defined (customized) exceptions rather than predefined exceptions. **It is highly recommended to define customized exceptions as unchecked exceptions** by extending RuntimeException instead of Exception.

In a custom exception, super("description") is used in the constructor to pass a description to the default exception handler, making the error message available when the exception is printed. **Example:**

class MyCustomException extends RuntimeException {

public MyCustomException(String message) {

super(message); // Pass message to RuntimeException

}

}

public class Test {

public static void main(String[] args) {

throw new MyCustomException("Custom exception occurred");

}

}

Output: Exception in thread "main" MyCustomException: Custom exception occurred

at Test.main(Test.java:9)

**→** In Java, exceptions can be broadly categorized into two types based on who raises the exception:

**JVM Exceptions:**

**-----------------------**

- Raised by the Java Virtual Machine (JVM)

- Typically occur due to environmental or system-related issues

- Examples:

- NullPointerException (when trying to access a null object reference)

- OutOfMemoryError (when the JVM runs out of memory)

- ClassCastException (when trying to cast an object to an incompatible type)

**Programmatic Exceptions:**

**------------------------------------**

- Raised by the application code itself

- Typically occur due to logical errors or invalid user input

- Examples:

- FileNotFoundException (when trying to access a file that does not exist)

- SQLException (when there's an error interacting with a database)

- Custom exceptions defined by the application (e.g., InvalidUserInputException)

Note that while JVM exceptions are typically unchecked (i.e., they don't need to be declared or caught), programmatic exceptions can be either checked or unchecked, depending on how they're defined. Here are the descriptions of the top 10 exceptions:

**ArrayIndexOutOfBoundsException:**

**------------------------------------------------**

Type: Unchecked (RuntimeException)

Category: JVM Exception

Cause: Raised when trying to access an array element with an invalid index (negative or beyond array size).

**NullPointerException:**

**------------------------------**

Type: Unchecked (RuntimeException)

Category: JVM Exception

Cause: Raised when accessing methods or fields of an object reference that is null.

**ClassCastException:**

**---------------------------**

Type: Unchecked (RuntimeException)

Category: JVM Exception

Cause: Raised when attempting to cast an object to an incompatible type at runtime.

**StackOverflowError:**

**----------------------------**

Type: Unchecked (Error)

Category: JVM Exception

Cause: Raised due to excessive recursive method calls leading to stack memory exhaustion.

**NoClassDefFoundError:**

**--------------------------------**

Type: Unchecked (Error)

Category: JVM Exception

Cause: Raised when a class is not found at runtime, though it was available during compilation.

**ExceptionInInitializerError:**

**------------------------------------**

Type: Unchecked (Error)

Category: JVM Exception

Cause: Raised when a static initializer block or static variable initialization fails due to an exception.

**IllegalArgumentException:**

**------------------------------------**

Type: Unchecked (RuntimeException)

Category: Programmatic Exception

Cause: Raised when an invalid argument is passed to a method.

**NumberFormatException:**

**------------------------------------**

Type: Unchecked (RuntimeException)

Category: Programmatic Exception

Cause: Raised when trying to convert an invalid string to a numeric format (e.g., Integer.parseInt("abc")).

**IllegalStateException:**

**-----------------------------**

Type: Unchecked (RuntimeException)

Category: Programmatic Exception

Cause: Raised when an operation is performed at an inappropriate state of an object.

**AssertionError:**

**---------------------**

Type: Unchecked (Error)

Category: Programmatic Exception

Cause: Raised when an assertion (assert statement) fails, typically used in debugging.

**→ Java 7 (JDK 1.7) introduced two significant enhancements to exception handling:**

**Try-with-resources:** This feature automates the closing of resources (like files, network connections, or database connections) that implement the AutoCloseable interface. It ensures that these resources are closed even if exceptions occur within the try block. This greatly simplifies resource management and prevents resource leaks.

**Multi-catch block:** This feature allows you to catch multiple different exception types in a single catch block. This reduces

code duplication when you have similar handling logic for multiple exceptions. It makes the code more concise and readable.

**→** Until Java 1.6, it was highly recommended to use the finally block to close resources opened within the try block. However, this approach required developers to manually close resources, increasing code complexity, length, and reducing readability. Example of using multiple resources in a try-with-resources statement:

import java.io.\*;

public class TryWithResourcesExample {

public static void main(String[] args) {

try (

FileReader fr = new FileReader("input.txt");

BufferedReader br = new BufferedReader(fr)

) {

System.out.println("First Line: " + br.readLine());

} catch (IOException e) {

e.printStackTrace();

}

}

}

To overcome this, try-with-resources was introduced **in Java 1.7. Its main advantage is that resources declared within the try block are automatically closed when the block exits**, regardless of whether execution completes normally or an exception occurs. This reduces complexity, shortens code, and improves readability.

**→** We can declare multiple resources in a try-with-resources statement, but they must be separated by a semicolon **(;)**. All resources used in this construct must be auto-closeable, meaning the corresponding class must implement the AutoCloseable interface. A resource is auto-closeable if it provides a close() method, which is automatically invoked when the try block completes execution.

All I/O, database, and network-related resources already implement AutoCloseable, so programmers don’t need to do anything extra.

**AutoCloseable was introduced in Java 1.7 and contains only one method: close().**

**→ All resource reference variables are implicitly final**, meaning reassignment within the try block is not allowed and will result in a compilation error. Until Java 1.6, a try block had to be associated with either a catch or finally block, but starting **from Java 1.7, try-with-resources can be used without catch or finally**.Here's an example demonstrating both points:

import java.io.\*;

public class TryWithResources {

public static void main(String[] args) {

try (FileReader fr = new FileReader("input.txt")) {

// fr = new FileReader("anotherfile.txt"); // Uncommenting this line will cause CE

System.out.println("Resource opened successfully.");

} // No need for catch or finally in Java 1.7+

}

}

(Explanation: If we try to reassign fr to another FileReader instance, the compiler will give an error. This is valid in Java 1.7+

because resources are automatically closed.)

**→** A multi-catch block allows handling multiple exceptions in a single catch block, introduced in Java 1.7. It improves code

readability and reduces duplication by using the | (pipe) operator to specify multiple exceptions. Example:

public class MultiCatchExample {

public static void main(String[] args) {

try {

int[] arr = new int[5];

arr[10] = 30 / 0; // May throw ArithmeticException or ArrayIndexOutOfBoundsException

} catch (ArithmeticException | ArrayIndexOutOfBoundsException e) {

System.out.println("Exception caught: " + e);

}

}

}

**( Key Points:** The pipe (|) operator is used to separate multiple exception types.The exception variable (e) is implicitly final,

so modification is not allowed inside the catch block. **The exceptions should not have a parent-child relationship, or else it**

**will cause a compilation error. )**

**→** The toString() method is overridden in the Throwable class. The overridden toString() method in Throwable is designed

to return a string that represents the exception, including its description (the message associated with the exception).

**Exception Propagation:** Exception propagation describes the process by which an exception is passed up the call stack to

the calling methods when it's not handled (using a try-catch block) in the method where it originated. When a method

throws an exception, the JVM searches for a suitable catch block to handle it. If the current method doesn't have a

matching catch block, the exception propagates to the method that called the current method, and the search continues up the call stack until either a catch block is found, or the exception reaches the top of the call stack (usually the main method), at which point the program might terminate abnormally.

**Re-throwing an Exception:** Re-throwing an exception occurs when a catch block catches an exception, performs some

action (e.g., logging, cleanup), and then re-throws the same exception or a different exception (often a wrapped exception).

This can be done using the throw keyword within the catch block. Re-throwing is useful when you want to handle part of

the exception processing at one level but want the exception to be further handled or propagated up the call stack for more

general handling. It is also frequently used to wrap a checked exception in an unchecked exception to avoid having to

declare throws everywhere.

**→** If there are multiple return statements in a try-catch-finally block (or even just in the try or catch blocks), and there's also

a return statement in the finally block, the return statement in the finally block always takes precedence. The value

returned will be the one from the finally block, even if a return statement was encountered earlier in the try or catch blocks.

The finally block's return essentially "overrides" any previous return values.

**→** The finally block is not guaranteed to execute if the program terminates abruptly due to a call to System.exit(). When

System.exit() is called (with any status code, including 0), the JVM begins its shutdown process. The JVM is responsible for

executing the finally block, but if the JVM itself is shutting down, it might not get a chance to do so. **Therefore, System.exit()**

**is the only reliable way to prevent a finally block from executing.**

**→** In Java, when an exception is raised in an inner try block and there is no matching catch block within the inner try block, the outer try block will attempt to catch the exception. If the outer try block has a catch block that matches the exception type, the exception will be caught and handled by the outer catch block. However, if the outer try block does not have a matching catch block, the exception will propagate up the call stack, leading to abnormal termination of the program.

**Java 9 Enhancement in Try-With-Resources:** In Java 7, when using try-with-resources, the resource must be declared inside

the try block. Example:

try (BufferedReader br = new BufferedReader(new FileReader("test.txt"))) {

System.out.println(br.readLine());

} catch (IOException e) {

e.printStackTrace();

}

Here, br is declared inside try(), so it is automatically closed.

**→ From Java 9 onwards**, we can use already declared final or effectively final resources in try-with-resources. No need to

declare resources inside try() explicitly. Example:

BufferedReader br = new BufferedReader(new FileReader("test.txt"));

try (br) {

System.out.println(br.readLine());

} catch (IOException e) {

e.printStackTrace();

}

Here, br is declared before try, and it is effectively final. It is automatically closed after try block execution.

**Advantages:** Reduces code duplication (no need to redeclare resource inside try()). Improves readability and maintainability.

No additional requirement to explicitly mark variables as final (as long as they are effectively final).

**Multithreading**

**====================================================================================**

**Multithreading is a feature in Java** that allows multiple threads to execute concurrently within a single program. A thread is

the smallest unit of a process, and multithreading enables a program to perform multiple tasks simultaneously, improving

performance and responsiveness.

Key Concepts of Multithreading:

--------------------------------------------

**Thread:** A lightweight sub-process that executes independently. Each thread has its own stack but shares the heap memory with other threads.

**Concurrency:** The ability of a program to execute multiple threads simultaneously.

**Parallelism:** The ability of a program to execute multiple threads in parallel(on multi-core processors)

**Thread Lifecycle:** A thread goes through various states: New, Runnable, Running, Blocked/Waiting, and Terminated.

Advantages of Multithreading:

-----------------------------------------

**Improved Performance:** Utilizes CPU resources efficiently by executing multiple tasks concurrently.

**Responsiveness:** Keeps the application responsive by performing background tasks in separate threads.

**Resource Sharing:** Threads share memory and resources, reducing overhead.

**Simplified Modeling:** Makes it easier to model real-world scenarios where multiple tasks occur simultaneously.

Creating Threads in Java:

----------------------------------

There are **two ways** to create threads in Java:

**Extending the Thread Class:** Create a subclass of Thread and override the run() method. Example:

class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

}

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // Starts the thread

}

}

**Implementing the Runnable Interface:** Implement the Runnable interface and pass it to a Thread object.

Thread Lifecycle:

-----------------------

**New:** The thread is created but not yet started.

**Runnable:** The thread is ready to run and waiting for CPU time.

**Running:** The thread is executing its task.

**Blocked/Waiting:** The thread is waiting for a resource or another thread.

**Terminated:** The thread has completed its task or been terminated.

**→ Thread Synchronization:** When multiple threads access shared resources, it can lead to data inconsistency. **Solution:** Use

synchronization to ensure that only one thread can access the shared resource at a time.

Real-World Use Cases:

-------------------------------

**Web Servers:** Handle multiple client requests simultaneously using threads.

**Games:** Run game logic, rendering, and input handling in separate threads.

**Data Processing:** Process large datasets in parallel using multiple threads.

(**Summary:** Multithreading allows a program to perform multiple tasks simultaneously. Threads can be created by extending

the Thread class or implementing the Runnable interface. Synchronization is used to prevent data inconsistency in

multithreaded programs.)

**→** Both process-based and thread-based multitasking are techniques that allow a computer to execute multiple tasks

seemingly at the same time. However, they differ in how they achieve this concurrency:

**Process-based multitasking:**

**---------------------------------------**

**Processes:** In process-based multitasking, the operating system manages multiple independent processes. Each process is a

separate instance of a program, with its own dedicated memory space and resources (like file handles).

**Concurrency:** The operating system rapidly switches between these processes, giving the illusion of simultaneous execution.

**Overhead:** Creating and managing processes has a higher overhead compared to threads, as each process requires its own

memory space and resources.

**Thread-based multitasking:**

**--------------------------------------**

**Threads:** In thread-based multitasking, multiple threads of execution operate within a single process. Threads share the

same memory space and resources of the process.

**Concurrency:** The operating system manages the execution of these threads, rapidly switching between them to create the

illusion of concurrency.

**Overhead:** Creating and managing threads has a lower overhead compared to processes, as threads share the same

resources.

**→** Multitasking, whether process-based or thread-based, is a technique used in operating systems to improve system

performance and responsiveness. The main objectives of multitasking are:

**Reduced response time:** By switching between multiple tasks or processes, the system can respond quickly to user input

and events, even if one task is taking a long time to complete.

**Improved system performance:** Multitasking allows the system to make efficient use of system resources, such as CPU,

memory, and I/O devices, by allocating them to multiple tasks concurrently.

**Increased throughput:** By executing multiple tasks simultaneously, the system can complete more tasks in a given time

period, increasing overall system throughput.

**→** Java makes it relatively easy to develop multithreaded applications compared to older languages. Here's why: Java's

Built-in Multithreading Support. Java provides a rich set of APIs and built-in support for multithreading, making it easier to

develop concurrent programs.

**→ The thread scheduler is a crucial part** of the operating system (or, in the case of Java, the JVM's implementation of the

operating system's thread management) that is responsible for deciding which thread should run at any given time. It's the

mechanism that manages the execution of multiple threads, giving each thread a slice of CPU time.

Here's a breakdown of its role:

-----------------------------------------

**Thread Management:** The thread scheduler keeps track of all the runnable threads in the system. It maintains a data structure (often a queue or a priority queue) of threads that are ready to run.

**CPU Time Allocation:** The scheduler's primary job is to allocate CPU time to the runnable threads. It determines which

thread gets to use the CPU and for how long.

**Scheduling Algorithms:** The thread scheduler uses various scheduling algorithms to make these decisions.

**Context Switching:** When the scheduler switches from one thread to another, it performs a context switch.

**→** In multithreaded programs, especially when dealing with shared resources and without proper synchronization, there's

generally no guarantee of a specific, predictable output. The exact output can vary from run to run due to the non-

deterministic nature of thread scheduling.

**→ Calling start()** on a thread creates a new thread that executes the run() method, whereas **calling run() directly** executes

it like a normal method in the main thread.

**→** The start() method in the Thread class is responsible for registering the thread with the thread scheduler and handling

other necessary setup tasks. Without calling start(), a new thread will not be created, making it a crucial method for

multithreading.

**→** Overloading the run() method is possible, but the start() method in the Thread class will always invoke the no-argument

run() method. Any other overloaded run() methods must be called explicitly like a normal method.

**→** If the run() method is not overridden, the Thread class's default run() method (which has an empty implementation) will

be executed, resulting in no output.

**→** It is highly recommended to override the run() method but not the start() method. If we override the start() method, it

will be treated as a normal method call, and no new thread will be created.

**→** If a thread has already been started and we try to restart the same thread, **it will throw a IllegalThreadStateException at**

**runtime.**

**→** The Runnable interface is present in the java.lang package and contains only **one method: void run();** It is a functional

interface and is commonly used for defining the task that a thread will execute.

Example: Implementing the Runnable Interface

class MyRunnable implements Runnable {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

}

}

}

public class Main {

public static void main(String[] args) {

MyRunnable myRunnable = new MyRunnable();

Thread t1 = new Thread(myRunnable);

t1.start(); // Starts the thread

}

}

Explanation of the Cases:

----------------------------------

MyRunnable r = new MyRunnable();

Thread t1 = new Thread();

Thread t2 = new Thread(r);

Let’s analyze each case:

1. **t1.start():**

**----------------**

What Happens: A new thread is created and started. The run() method of the Thread class is executed, which does nothing by default.

Output: No output, as the default run() method of Thread is empty.

1. **t1.run():**

**---------------**

What Happens: The run() method of the Thread class is executed in the current thread (not a new thread). No new thread is

created.

Output: No output, as the default run() method of Thread is empty.

1. **t2.start():**

**-----------------**

What Happens: A new thread is created and started. The run() method of the MyRunnable class (passed to t2) is executed in

the new thread.

Output: Depends on the implementation of MyRunnable's run() method.

1. **t2.run():**

**---------------**

What Happens: The run() method of the MyRunnable class is executed in the current thread (not a new thread). No new

thread is created.

Output: Depends on the implementation of MyRunnable's run() method.

1. **r.start():**

**---------------**

What Happens: Compilation Error (CE). MyRunnable does not have a start() method because it implements Runnable, not

Thread.

1. **r.run():**

**-------------**

What Happens: The run() method of the MyRunnable class is executed in the current thread (not a new thread). No new

thread is created.

Output: Depends on the implementation of MyRunnable's run() method.

**→** Implementing the Runnable interface is recommended over extending the Thread class because:

Preserves Inheritance **→** Since Java supports single inheritance, extending Thread restricts the ability to extend another

class, whereas implementing Runnable allows the class to extend other classes if needed.

Better Separation of Concerns **→** The Runnable interface focuses only on defining the task, while Thread handles thread management, making the code more flexible and reusable.

Encourages Code Reusability **→** The same Runnable instance can be used with multiple threads, whereas extending Thread

binds both task and thread together.

**→ The Thread class in Java provides 8 constructors**, which allow different ways to create a thread. Here are their

signatures:

// 1. Default constructor

public Thread()

// 2. Constructor with Runnable target

public Thread(Runnable target)

// 3. Constructor with Thread name

public Thread(String name)

// 4. Constructor with Runnable target and Thread name

public Thread(Runnable target, String name)

// 5. Constructor with ThreadGroup and Runnable target

public Thread(ThreadGroup group, Runnable target)

// 6. Constructor with ThreadGroup, Runnable target, and Thread name

public Thread(ThreadGroup group, Runnable target, String name)

// 7. Constructor with ThreadGroup, Runnable target, Thread name, and stack size

public Thread(ThreadGroup group, Runnable target, String name, long stackSize)

// 8. Constructor with ThreadGroup and Thread name

public Thread(ThreadGroup group, String name)

( These constructors provide flexibility to create threads with different configurations, such as

associating them with a ThreadGroup, setting their name, and defining a stack size.)

**Hybrid Approach: Combining Thread and Runnable:**

**-----------------------------------------------------------------------**

**In this approach:** You create a class that extends Thread (e.g., MyThread). You pass an instance of this class to a Thread

constructor (e.g., Thread t2 = new Thread(t1)). You call t2.start() to start the thread.

**This works because:** The Thread class itself implements the Runnable interface. When you pass a Thread object (e.g., t1) to

another Thread constructor (e.g., t2), the run() method of the Thread object (t1) will be executed when t2.start() is called.

Example of Hybrid Approach:

class MyThread extends Thread {

@Override

public void run() {

System.out.println("MyThread is running: " + Thread.currentThread().getName());

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread(); // t1 is a Thread object

Thread t2 = new Thread(t1); // Pass t1 to another Thread constructor

t2.start(); // Start the thread

}

}

Output: MyThread is running: Thread-0

**→ Every thread in Java has a name**, either a default name assigned by the JVM (like "Thread-0", "Thread-1", etc.) or a

custom name set by the programmer. To get and set the name of a thread, we use the following two methods from the

Thread class:

public final String getName() **→** Retrieves the name of the thread.

public final void setName(String name) **→** Sets a custom name for the thread.

(These methods help in identifying and managing threads efficiently in multi-threaded applications.)

**→** We can get the current executing thread object by using the **Thread.currentThread() method**. It is a static method

present in the Thread class. It returns a reference to the currently executing thread. It is useful for getting details like the

thread's name, priority, and state. Example:

Thread t = Thread.currentThread();

System.out.println("Current Thread: " + t.getName());

**→** Every thread in Java has some priority, it may be the default priority assigned by the JVM or a customized priority set by

the programmer. The valid range of thread priority is 1 to 10, where 1 is the minimum priority, and 10 is the maximum

priority. Constants defined in the Thread class are:

Thread.MIN\_PRIORITY → 1 (Lowest priority)

Thread.NORM\_PRIORITY → 5 (Default priority)

Thread.MAX\_PRIORITY → 10 (Highest priority)

**Important Points:**

**Default Priority:** If we do not set a priority, the thread gets **NORM\_PRIORITY** **(5)** by default.

**Setting Priority:** We can change a thread’s priority using **setPriority(int priority).**

**Getting Priority:** We can check the thread’s priority using **getPriority().**

**Thread Scheduler:** The thread with a higher priority **does not always execute first**. It depends on the OS thread scheduler.

Example:

Thread t = new Thread();

System.out.println("Default Priority: " + t.getPriority()); // Output: 5

t.setPriority(Thread.MAX\_PRIORITY);

System.out.println("Updated Priority: " + t.getPriority()); // Output: 10

**→** The thread scheduler uses priorities to determine which thread should be executed next. The thread with the highest

priority is given the chance to execute first. When two or more threads have the same priority, the thread scheduler (OS or

JVM) makes the ultimate decision about which thread gets to run and in what order.

**→** Thread class defines the following methods to get and set the priority of a thread:

public final int getPriority() → Retrieves the priority of the thread.

public final void setPriority(int priority) → Sets the priority of the thread.

**→ The allowed values range from 1 to 10**, if an invalid value is provided, it will throw **IllegalArgumentException** at runtime.

**→** The default priority for the main thread is 5. **For all other threads, the default priority is inherited from the parent thread.** This means that if a parent thread has a certain priority, any child thread created by it will have the same priority as the parent. Example:

class MyThread extends Thread {

public void run() {

System.out.println("Child Thread Priority: " + this.getPriority());

}

}

public class ThreadPriorityExample {

public static void main(String[] args) {

System.out.println("Main Thread Default Priority: " + Thread.currentThread().getPriority());

// Changing the priority of the main thread

Thread.currentThread().setPriority(7);

System.out.println("Main Thread Updated Priority: " + Thread.currentThread().getPriority());

// Creating a child thread

MyThread t = new MyThread();

t.start(); // Child thread will inherit the priority of the main thread

}

}

**Output:**

Main Thread Default Priority: 5

Main Thread Updated Priority: 7

Child Thread Priority: 7

**→** Some platforms do not provide proper support for thread priorities, meaning a thread with a higher priority may not always execute first. Thread scheduling depends on the underlying OS and JVM implementation, so priority-based execution is not guaranteed across all environments.

**→** In Java, you can control or prevent a thread's execution using the following methods: **yield(), join(), sleep()**

**yield():**

**----------**

Purpose: The yield() method pauses the current thread and gives other threads of the same or higher priority a chance to execute. It is a hint to the thread scheduler, **but there is no guarantee that the thread will actually pause.**

Use Case: Used to improve the fairness of thread execution, allowing other threads to run.

**→** The thread that is yielded will get the chance to execute again depending on the Thread Scheduler, and we cannot predict exactly when it will run.

The complete prototype of the yield() method is: **public static native void yield();**

When yield() is called, the thread moves from the **running state to the runnable state**, allowing other threads of the same or higher priority a chance to execute. Example:

class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

Thread.yield(); // Pause the current thread

}

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

MyThread t2 = new MyThread();

t1.start();

t2.start();

}

}

**Output (may vary):**

Thread-0: 1

Thread-1: 1

Thread-0: 2

Thread-1: 2

Thread-0: 3

Thread-1: 3

**join():**

**---------**

Purpose: The join() method waits for a thread to die (i.e., complete its execution) before continuing the current thread. It is useful when one thread depends on the result of another thread.

Use Case: Used to ensure that a thread completes its task before another thread starts.

**→** If there are two threads, t1 and t2, and t1 depends on the result of t2, then t1 should call t2.join(). This causes t1 to enter the waiting state until t2 completes its execution.

**→** The Thread class provides three overloaded join() methods:

**public final void join() throws InterruptedException:** The calling thread waits indefinitely until the target thread completes.

**public final void join(long millis) throws InterruptedException:** The calling thread waits for the specified milliseconds for the target thread to complete.

**public final void join(long millis, int nanos) throws InterruptedException:** The calling thread waits for the specified milliseconds and nanoseconds for the target thread to complete.

**Impact of join() Method on Thread Lifecycle:**

**-------------------------------------------------------------**

When a thread T1 calls T2.join(), T1 enters the waiting state and pauses execution until T2 completes. This affects the thread lifecycle as follows:

T1 moves to the waiting state (blocked from execution). T2 continues execution until it completes (reaches the terminated

state). Once T2 finishes, T1 moves from waiting to runnable state and is eligible for execution again.

A thread in the waiting state due to join() can return to runnable state in the following cases:

**T2 completes execution →** T1 resumes execution.

**T1 calls join(millis) or join(millis, nanos), and the specified time expires →** T1 resumes execution.

**Another thread interrupts T1 while waiting (causing InterruptedException) →** T1 resumes execution.

**Example 1:**

class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

}

}

}

public class Main {

public static void main(String[] args) throws InterruptedException {

MyThread t1 = new MyThread();

t1.start();

t1.join(); // Wait for t1 to complete

System.out.println("Main thread continues after t1 completes.");

}

}

Output:

Thread-0: 1

Thread-0: 2

Thread-0: 3

Main thread continues after t1 completes.

**Example 2:**

public class Test {

public static void main(String[] args) throws InterruptedException {

MyThread.mt = Thread.currentThread();

MyThread t1 = new MyThread();

t1.start();

for (int i = 1; i <= 5; i++) {

System.out.println("Main Thread - " + i);

Thread.sleep(1000);

}

}

}

class MyThread extends Thread {

static Thread mt;

@Override

public void run() {

try {

mt.join();

} catch (InterruptedException e) {

e.getMessage();

}

for (int i = 1; i <= 5; i++) {

System.out.println("Child Thread - "+ i);

}

}

}

In the given example, the child thread (t1) calls mt.join(); inside its run() method. Since mt refers to the main thread, the child thread waits until the main thread completes execution.

**Output Behavior:** The main thread prints first (1 to 5). Only after the main thread completes, the child thread starts printing (1 to 5).

**→** If the main thread calls t1.join(); (waiting for the child thread) and the child thread calls mt.join(); (waiting for the main thread), both threads will be waiting for each other indefinitely, **leading to a deadlock situation where neither thread can proceed.**

**→** If a thread calls join() on itself, it will enter a waiting state forever, leading to a deadlock-like situation where the program will never proceed. **Example of Self-Join Deadlock:**

public class Test {

public static void main(String[] args) throws InterruptedException {

Thread.currentThread().join(); // Main thread waiting for itself

System.out.println("This line will never execute.");

}

}

**sleep():**

**----------**

Purpose: The sleep() method pauses the current thread for a specified amount of time (in milliseconds). It is useful for introducing delays or simulating time-consuming tasks.

Use Case: Used to pause a thread for a specific duration.

**→** The Thread class provides the following two prototypes of the sleep() method:

sleep(long millis):  **public static native void sleep(long millis) throws InterruptedException;**

Puts the current thread to sleep for the specified time in milliseconds. The thread will be in the TIMED\_WAITING state for at least the given time. After sleep time ends, the thread goes to the Runnable state.

sleep(long millis, int nanos):

**public static void sleep(long millis, int nanos) throws InterruptedException {**

// Logic to handle nanos and call native sleep

**}**

**(**The sleep(long millis, int nanos) method is not native; it is implemented in Java but internally calls the native sleep() method.**)**

Puts the current thread to sleep for the specified time in milliseconds and nanoseconds.

millis: The number of milliseconds to sleep.

nanos: The number of extra nanoseconds (0 to 999999).

This provides more precise control over sleeping time.

**Key Points:** sleep() is a static method in Thread class. The thread does not release locks while sleeping. If interrupted during sleep, it throws InterruptedException.

**→** A thread moves from the sleeping state to the runnable state in the following two cases:

**When the specified sleep duration expires:** If a thread is put to sleep using Thread.sleep(time), it will automatically wake up and move to the runnable state once the specified time has elapsed.

**When the sleeping thread is interrupted:** If another thread calls interrupt() on a sleeping thread, the sleeping thread immediately moves to the runnable state and throws an InterruptedException.

Example:

class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 5; i++) {

System.out.println(Thread.currentThread().getName() + ": " + i);

try {

Thread.sleep(1000); // Pause for 1 second

} catch (InterruptedException e) {

System.out.println("Thread interrupted.");

}

}

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start();

}

}

Output (with 1-second delay between each line):

Thread-0: 1

Thread-0: 2

Thread-0: 3

Thread-0: 4

Thread-0: 5

**→** A thread can interrupt a sleeping or waiting thread using the interrupt() method of the Thread class. Prototype of interrupt() method: **public void interrupt();**

When a thread is sleeping or waiting, calling interrupt() will force it to wake up and throw an InterruptedException. Simple

Example: Interrupting a Sleeping Thread

class MyThread extends Thread {

@Override

public void run() {

try {

System.out.println("Thread going to sleep...");

Thread.sleep(5000); // Thread enters sleeping state

System.out.println("Thread woke up normally.");

} catch (InterruptedException e) {

System.out.println("Thread was interrupted!");

}

}

}

public class Test {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start();

try {

Thread.sleep(2000); // Main thread sleeps for 2 seconds

t.interrupt(); // Interrupting the sleeping thread

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

Output:

-----------

Thread going to sleep...

Thread was interrupted!

( Here, t is put to sleep for 5 seconds. After 2 seconds, the main thread calls t.interrupt(), which wakes up t and throws an InterruptedException. Instead of completing its sleep, the thread immediately moves to the runnable state and prints "Thread was interrupted!".)

**→** If we call the interrupt() method on a thread that is not in a sleeping or waiting state, then the interrupt call has no immediate effect. However, if the target thread later enters the sleeping or waiting state, it will be interrupted at that moment. If the thread never enters a sleeping or waiting state, the interrupt call will have no effect at all. This is the only case where the interrupt() call is wasted. Example: Interrupt Call with No Effect:

class MyThread extends Thread {

@Override

public void run() {

for (int i = 1; i <= 2; i++) {

System.out.println("Thread executing: " + i);

}

System.out.println("Thread completed without sleep/wait.");

}

}

public class Test {

public static void main(String[] args) {

MyThread t = new MyThread();

t.start();

t.interrupt(); // Interrupt call has no effect

}

}

Output:

Thread executing: 1

Thread executing: 2

( Thread completed without sleep/wait. Here, interrupt() is called on t, but since the thread never enters sleep or wait, the interrupt call has no impact.)

**Key Differences Between yield(), join(), and sleep():**

**Method Purpose Thread State InterruptedException Use Case**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

yield() Pauses the current Running → Runnable No Improve fairness in thread to give other threads thread execution.

a chance to execute.

join() Waits for a thread to execution. Running → Waiting Yes Ensure thread completion

complete its before proceeding.

sleep() Pauses the current thread Running → Timed Waiting Yes Introduce delays or simulate tasks.

for a specified amount of time.

**→ The synchronized keyword** is used to handle concurrent access to shared resources. It can be applied to methods and blocks, but not to classes or variables.

Key Points:

---------------

**Ensures thread safety:** Prevents multiple threads from accessing a synchronized method/block on the same object simultaneously.

**Resolves data inconsistency:** Only one thread at a time can execute a synchronized method/block, ensuring consistency in shared data.

Two types of synchronization:

----------------------------------------

**Method-level synchronization** (synchronized method)

**Block-level synchronization** (synchronized block) → more efficient as it locks only a portion of the code instead of the entire method.

**→** The main disadvantage of the synchronized keyword is that it can cause performance issues due to increased thread waiting time. When one thread holds the lock, other threads must wait, leading to potential thread contention and reduced concurrency.

Key Points:

----------------

**Increased waiting time →** Only one thread can execute a synchronized method/block at a time, making other threads wait.

**Performance overhead →** Context switching and thread scheduling overhead can degrade performance.

**Deadlocks possibility →** Improper synchronization can lead to deadlocks.

**Best Practice:** Use synchronization only when necessary. Alternatives like Concurrent API (e.g., ReentrantLock, Atomic variables) can be considered for better performance.

**→** Synchronization in Java is implemented using locks, and every Java object has a unique intrinsic lock (monitor lock).

Key Points on Synchronization & Locks:

-----------------------------------------------------

Lock Acquisition **→** When a thread wants to execute a synchronized method/block, it must first acquire the lock of the object.

Lock Release **→** The lock is automatically released when the thread exits the synchronized method/block, either normally or due to an exception.

One Thread at a Time **→** While a thread holds the lock, no other thread can execute any synchronized method on the same object.

JVM Handles Locking **→** The process of acquiring and releasing the lock is handled internally by the JVM, so the programmer does not need to manage it manually.

Example:

class SharedResource {

synchronized void display() { // Only one thread can execute this at a time

System.out.println(Thread.currentThread().getName() + " has acquired the lock.");

try { Thread.sleep(1000); } catch (InterruptedException e) {}

System.out.println(Thread.currentThread().getName() + " is releasing the lock.");

}

}

class MyThread extends Thread {

SharedResource resource;

MyThread(SharedResource resource) { this.resource = resource; }

public void run() { resource.display(); }

}

public class SyncExample {

public static void main(String[] args) {

SharedResource obj = new SharedResource();

Thread t1 = new MyThread(obj);

Thread t2 = new MyThread(obj);

t1.start();

t2.start();

}

}

Output: (Order may vary)

Thread-0 has acquired the lock.

Thread-0 is releasing the lock.

Thread-1 has acquired the lock.

Thread-1 is releasing the lock.

Since the display() method is synchronized, one thread acquires the lock, executes the method, and releases the lock before another thread can enter.

**→** While a thread is executing a synchronized method on an object, other threads cannot execute any synchronized method on the same object at the same time. However, other threads can still execute non-synchronized methods on that object simultaneously.

**Key Points:** Synchronized methods block other synchronized methods on the same object. Non-synchronized methods do not require the lock, so they can run in parallel with synchronized methods. Each object has a single lock; locking applies only to synchronized methods.

**→** Locks in Java are fundamentally associated with objects. The synchronized keyword (whether used on a method or a block) operates on this object-level lock. It's generally recommended to keep the minimum amount of code within the synchronized block or method. This reduces the time that the lock is held, improving concurrency.

**Synchronized area:** You should put the code that modifies the shared state of the object (the "critical section") within the synchronized block or method. This ensures that only one thread can modify the shared state at a time, preventing race conditions and data corruption.

**Non-synchronized area:** Code that reads the shared state or performs operations that don't modify the state can be outside the synchronized block. Multiple threads can safely read shared data concurrently without synchronization issues. Keeping read operations outside the synchronized block increases concurrency and performance.

**→** Synchronization is only required when multiple threads are operating on the same Java object to prevent data inconsistency. If multiple threads are operating on different Java objects, each object has its own lock, so synchronization is not needed because they don’t interfere with each other.

**→** In Java, every class has a unique lock, which is used for class-level synchronization (i.e., when synchronizing static

methods). If a thread wants to execute a static synchronized method, it needs to **acquire the class-level lock.** Once it gets the class lock, **no other thread can execute any other static synchronized method of that class simultaneously.** However, other threads can still execute:

Normal static methods (since they don’t require synchronization). **Synchronized instance methods** (because they require an **instance lock**, not the class lock). Normal instance methods (since they are not synchronized).

Example: Class-Level Lock with Static Synchronized Methods

class MyClass {

synchronized static void staticSyncMethod1() {

System.out.println(Thread.currentThread().getName() + " got class lock - staticSyncMethod1");

try { Thread.sleep(2000); } catch (InterruptedException e) {}

System.out.println(Thread.currentThread().getName() + " finished staticSyncMethod1");

}

synchronized static void staticSyncMethod2() {

System.out.println(Thread.currentThread().getName() + " got class lock - staticSyncMethod2");

try { Thread.sleep(2000); } catch (InterruptedException e) {}

System.out.println(Thread.currentThread().getName() + " finished staticSyncMethod2");

}

static void normalStaticMethod() {

System.out.println(Thread.currentThread().getName() + " executing normalStaticMethod");

}

synchronized void instanceSyncMethod() {

System.out.println(Thread.currentThread().getName() + " executing instanceSyncMethod");

}

void normalInstanceMethod() {

System.out.println(Thread.currentThread().getName() + " executing normalInstanceMethod");

}

}

public class ClassLockExample {

public static void main(String[] args) {

MyClass obj1 = new MyClass();

MyClass obj2 = new MyClass();

Thread t1 = new Thread(() -> MyClass.staticSyncMethod1(), "Thread-1");

Thread t2 = new Thread(() -> MyClass.staticSyncMethod2(), "Thread-2");

Thread t3 = new Thread(() -> MyClass.normalStaticMethod(), "Thread-3");

Thread t4 = new Thread(() -> obj1.instanceSyncMethod(), "Thread-4");

Thread t5 = new Thread(() -> obj2.normalInstanceMethod(), "Thread-5");

t1.start();

t2.start();

t3.start();

t4.start();

t5.start();

}

}

Expected Output (Order May Vary):

------------------------------------------------

Thread-1 got class lock - staticSyncMethod1

Thread-3 executing normalStaticMethod

Thread-4 executing instanceSyncMethod

Thread-5 executing normalInstanceMethod

(Thread-1 completes after 2 sec)

Thread-1 finished staticSyncMethod1

Thread-2 got class lock - staticSyncMethod2

(Thread-2 completes after 2 sec)

Thread-2 finished staticSyncMethod2

Explanation:

------------------

Thread-1 starts executing staticSyncMethod1(), locking the class.

Thread-2 has to wait for Thread-1 to release the class-level lock before it can execute staticSyncMethod2().

Thread-3 executes normalStaticMethod() without waiting (because it's not synchronized).

Thread-4 executes instanceSyncMethod() without waiting (because it's synchronized at the instance level,not the class level).

Thread-5 executes normalInstanceMethod() without waiting (since it's not synchronized).

**Synchronized Block in Java:**

**-------------------------------------**

A synchronized block is a block of code that is synchronized on a specific object, ensuring that only one thread can execute that block at a time for the given object.

Syntax of Synchronized Block:

synchronized (lockObject) {

// Critical section (code that needs synchronization)

}

Here, lockObject can be:

This reference (this) → for synchronizing instance methods.

A specific object → for finer control over synchronization.

Class-level lock (ClassName.class) → for synchronizing static methods.

Advantages of Synchronized Block:

-----------------------------------------------

**More Efficient Than Synchronized Methods:** Instead of synchronizing the entire method, we synchronize only the critical section that modifies shared data. This improves performance by allowing non-critical operations to run without blocking.

**Reduces Lock Contention:** If multiple threads are working on different parts of a method, they don't need to wait unnecessarily. Only the shared resource is locked, not the entire method.

**Provides More Granular Control:** You can synchronize different sections of code on different locks, rather than synchronizing the entire method. Helps avoid unnecessary blocking of unrelated code.

Example with Synchronized Block (Only Critical Section is Locked):

class Counter {

private int count = 0;

private final Object lock = new Object(); // Lock object

public void increment() {

synchronized (lock) { // Only this block is synchronized

count++;

}

}

}

Synchronized Block for Class-Level Lock (Static Context):

class MyClass {

public static void staticMethod() {

synchronized (MyClass.class) { // Class-level lock

System.out.println("Executing static synchronized block...");

}

}

}

This ensures that only one thread can execute the synchronized block in the static method at a time.

**→** The lock concept applies only to object types and class types, not primitive types like int, float, char, etc. Locks are

associated with objects, and primitive types are not objects. If you try to pass a primitive type to a synchronized block, the compiler will throw a Compile-Time Error (CE).

Example: Invalid Synchronized Block (Compile-Time Error)

class Test {

public void method() {

int x = 10;

synchronized (x) { // Compile-Time Error: Primitive types cannot be used

System.out.println("Inside synchronized block");

}

}

}

Error: unexpected type; required: reference, found: int

**(**Primitive types cannot be used as locks in a synchronized block. Instead, use object references (e.g., Integer, String, Object). Always synchronize on a shared object, not a local variable, to ensure proper thread safety.**)**

**→** In Java, when multiple threads access and modify the same object simultaneously, it can lead to data inconsistency issues, known as race conditions.

**What is a Race Condition?**

**------------------------------------**

A race condition occurs when the outcome of a program depends on the relative timing of threads or processes. In other words, the behavior of the program changes depending on which thread executes its code first.

**→** **A thread can acquire multiple locks simultaneously on different objects and also a class-level lock in Java.**

**How It Works? :** A thread can hold multiple object locks at the same time if it enters multiple synchronized blocks/methods on different objects. A thread can also acquire a class-level lock along with object locks if it executes a static synchronized method while holding instance locks. **Acquiring multiple locks is useful in nested synchronized blocks** but must be handled carefully to avoid deadlocks.

**→** The statements within a synchronized block or a synchronized method are called **synchronized statements (or sometimes the critical section).** These synchronized statements are designed to be executed by only one thread at a time.

**wait(), notify(), and notifyAll() Methods in Java:**

**-----------------------------------------------------------------**

These methods are used for **inter-thread communication** in Java. They allow threads to coordinate their activities by waiting for a condition to be met and notifying other threads when the condition is satisfied. These methods are part of the **Object class** and are used in conjunction with synchronized blocks or methods.

Key Concepts:

-------------------

**wait():** Causes the current thread to wait until another thread calls notify() or notifyAll() on the same object. **The thread releases the lock on the object immediately and enters the waiting state.**

**notify():** Wakes up a single thread that is waiting on the object's monitor. The awakened thread competes for the lock and resumes execution.

**notifyAll():** Wakes up all threads that are waiting on the object's monitor. The awakened threads compete for the lock and resume execution.

**How They Work:** These methods must be called from a synchronized context (i.e., inside a synchronized block or method). If these methods are called outside a synchronized context, a **IllegalMonitorStateException** will be thrown. **The thread calling wait() must hold the lock on the object. The thread calling notify() or notifyAll() must also hold the lock on the object.**

**→** The wait(), notify(), and notifyAll() methods are present in the Object class and not in the Thread class because: Every Java object has a monitor lock that can be used for synchronization. **A thread can call these methods on any Java object, not just on Thread instances.**

**→** When a thread calls notify() on an object, it does not release the lock immediately. The lock is released only after the

synchronized block or method completes execution. **wait(), notify(), and notifyAll() are the only methods** where a thread

releases the lock (when used correctly inside a synchronized block). Here are their prototypes:

wait() Methods:

---------------------

public final void wait() throws InterruptedException

public final native void wait(long timeout) throws InterruptedException;

public final void wait(long timeout, int nanos) throws InterruptedException

**Non-native wrapper:** The parameterless wait() and wait(long timeout, int nanos) are implemented in Java but internally call the native wait(long timeout).

notify() and notifyAll():

-------------------------------

public final native void notify();

public final native void notifyAll();

**→** When a thread enters the waiting state after calling wait(), it does not immediately proceed to runnable after being notified. Instead, it follows these steps:

**Lifecycle after wait() Call:**

**----------------------------------**

**Thread calls wait() →** It enters the waiting state and releases the lock of the object.

**Thread gets notified (notify() / notifyAll()) or times out (wait(timeout)):** If it gets notified, it moves to another waiting state where it waits to acquire the lock. If it times out, it also moves to this state. If it gets interrupted, an InterruptedException is thrown, and it exits waiting.

**Waiting to acquire the lock →** Since notify() does not release the lock immediately, the thread must wait until the notifying thread completes execution and releases the lock.

**Thread acquires the lock →** Now it moves to the runnable state.

**Thread scheduler picks the thread →** It enters the running state.

**Example: Producer-Consumer Problem:**

**-----------------------------------------------------**

The Producer-Consumer problem is a classic example of inter-thread communication. Here, a producer produces data, and a consumer consumes it. The producer and consumer must coordinate to ensure that the consumer does not consume data before it is produced. Code:

import java.util.LinkedList;

import java.util.Queue;

class SharedResource {

private final Queue<Integer> queue = new LinkedList<>();

private final int CAPACITY = 2;

// Producer method

public synchronized void produce(int value) throws InterruptedException {

while (queue.size() == CAPACITY) {

System.out.println("Queue is full. Producer is waiting...");

wait(); // Wait for the consumer to consume

}

queue.add(value);

System.out.println("Produced: " + value);

notify(); // Notify the consumer

}

// Consumer method

public synchronized void consume() throws InterruptedException {

while (queue.isEmpty()) {

System.out.println("Queue is empty. Consumer is waiting...");

wait(); // Wait for the producer to produce

}

int value = queue.poll();

System.out.println("Consumed: " + value);

notify(); // Notify the producer

}

}

public class Main {

public static void main(String[] args) {

SharedResource resource = new SharedResource();

// Producer thread

Thread producer = new Thread(() -> {

try {

for (int i = 1; i <= 5; i++) {

resource.produce(i);

Thread.sleep(1000); // Simulate production time

}

} catch (InterruptedException e) {

System.out.println("Producer interrupted.");

}

});

// Consumer thread

Thread consumer = new Thread(() -> {

try {

for (int i = 1; i <= 5; i++) {

resource.consume();

Thread.sleep(1000); // Simulate consumption time

}

} catch (InterruptedException e) {

System.out.println("Consumer interrupted.");

}

});

producer.start();

consumer.start();

}

}

Output:

-----------

Produced: 1

Consumed: 1

Produced: 2

Consumed: 2

Produced: 3

Consumed: 3

Produced: 4

Consumed: 4

Produced: 5

Consumed: 5

**Difference Between notify() and notifyAll():**

**-----------------------------------------------------------**

Both notify() and notifyAll() are used to wake up threads that are waiting on an object's monitor (i.e., threads that have called wait()). However, they differ in how many threads they wake up and their typical use cases.

**notify():**

**-----------**

**Purpose:** Wakes up a single thread that is waiting on the object's monitor. The choice of which thread to wake up is non deterministic (depends on the JVM implementation).

**Use Case:** Used when only one thread needs to be notified to proceed.

**Example:** In a producer-consumer scenario, if only one consumer needs to be notified to consume an item.

**Behavior:** If multiple threads are waiting, only one is awakened. The awakened thread must reacquire the lock before it can proceed.

**notifyAll():**

**---------------**

**Purpose:** Wakes up all threads that are waiting on the object's monitor. All awakened threads compete for the lock, and one of them will acquire it.

**Use Case:** Used when all waiting threads need to be notified to proceed.

**Example:** In a producer-consumer scenario, if multiple consumers need to be notified to consume items.

**Behavior:** All waiting threads are awakened. Only one thread can acquire the lock at a time, so the others will go back to waiting after being awakened.

**→ A deadlock occurs when two or more threads are blocked indefinitely**, waiting for each other to release resources that they need. This creates a circular dependency, preventing any of the threads from proceeding.

**synchronized and Deadlocks:** While the synchronized keyword (and other locking mechanisms) is a common cause of deadlocks. However, improper use of synchronized is a frequent culprit. Once a deadlock has happened, the threads involved are stuck. The JVM cannot automatically "fix" it. While detection and recovery mechanisms exist, they are not always reliable.

**Prevention Techniques:** The focus is on preventing deadlocks from occurring in the first place. This involves careful design and coding practices.

Example of deadlock:

-----------------------------

public class Test {

public static void main(String[] args) throws InterruptedException {

MyThread t1 = new MyThread();

t1.method();

}

}

class MyThread extends Thread {

A a = new A();

B b = new B();

void method() {

this.start();

a.m1(b);

}

@Override

public void run() {

b.m2(a);

}

}

class A {

synchronized void m1(B b) {

System.out.println("Inside A.m1()");

System.out.println("Going to call B's last method");

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.getMessage();

}

b.last();

System.out.println("End of A.m1() method");

}

synchronized void last() {

System.out.println("A.last() Method");

}

}

class B {

synchronized void m2(A a) {

System.out.println("Inside B.m2()");

System.out.println("Going to call A's last method");

try {

Thread.sleep(3000);

} catch (InterruptedException e) {

e.getMessage();

}

a.last();

System.out.println("End of B.m2() method");

}

synchronized void last() {

System.out.println("B.last() Method");

}

} // The synchronized keyword is the sole reason for the deadlock in this program. The deadlock occurs because:

**Thread-1 (main thread) calls a.m1(b):** This method is synchronized, so it acquires A's lock. Inside m1(), it sleeps for 3 seconds and then tries to call b.last(), requiring B's lock.

**Thread-2 (this.start() calls run(), which calls b.m2(a)):** m2() is synchronized, so it acquires B's lock. Inside m2(), it sleeps for 3 seconds and then tries to call a.last(), requiring A's lock.

**Deadlock Situation:** Thread-1 holds A's lock and is waiting for B's lock. Thread-2 holds B's lock and is waiting for A's lock. Since both threads are waiting on each other to release locks, they are stuck in a deadlock.

How to Prevent Deadlock?

------------------------------------

Remove at least one synchronized modifier (from either m1(), m2(), or last() methods). This prevents strict locking and allows execution to proceed. Careful lock ordering and alternative synchronization mechanisms can help prevent deadlocks.

**→ Starvation:** A thread experiences long waiting but eventually gets CPU time when higher-priority threads finish. Example of Starvation:

A low-priority thread may starve if high-priority threads keep executing continuously.However, if the high-priority threads eventually finish or yield, the low-priority thread gets a chance to execute. **( Deadlock = Waiting Never Ends, Starvation = Long Waiting But Ends Eventually )**

**→ Daemon threads** in Java are background threads that provide services to user threads. Here are the key characteristics:

Daemon threads are **low-priority threads** that run in the background. **The JVM will exit when all user threads are complete, even if daemon threads are still running.** Daemon threads are automatically terminated when the application finishes. They are typically used for background tasks like garbage collection, housekeeping operations, or monitoring services.

A thread can be set as a daemon thread before it starts using the **setDaemon(true)** method. **Daemon status is inherited from the parent thread.** Example usage:

javaCopyThread daemonThread = new Thread(() -> {

// Background task code here

});

daemonThread.setDaemon(true); // Must be called before starting the thread

daemonThread.start();

( Daemon threads are useful when you need background processing that should not prevent the application from terminating when all foreground tasks are complete.)

**→** Daemon Threads are background threads, usually with low priority, but we can set high priority if needed. We can check if a thread is daemon or not, **using isDaemon().** Once a thread is started, changing its daemon nature results in **IllegalThreadStateException.** Main Thread is always non-daemon, and **cannot be converted** into a daemon thread.

Example:

public class Test {

public static void main(String[] args) {

System.out.println(Thread.currentThread().isDaemon()); // false

// Thread.currentThread().setDaemon(true); RE: IllegalThreadStateException

MyThread t1 = new MyThread();

System.out.println(t1.isDaemon()); // false

t1.setDaemon(true);

t1.start();

System.out.println(t1.isDaemon()); // true

}

}

class MyThread extends Thread {}

**Java Multithreading Models:**

**---------------------------------------**

Green Thread Model (**Deprecated**, JVM-managed, no OS support)

Native OS Thread Model (**JVM + OS-managed**, widely used)

**Green Thread Model:** JVM manages threads without OS help. Limited support in modern OS. Deprecated and not recommended.

**Native OS Thread Model:** JVM depends on OS for thread management. Used in all modern Java versions. Windows, Linux, macOS support it. Java now fully relies on the Native OS Model for multithreading.

**Thread.stop():**

**-------------------**

Forcibly terminates thread execution. Thread enters dead state immediately. Deprecated because it's unsafe - releases all locked monitors, potentially leaving objects in inconsistent states. **Prototype: public void stop()**

Alternative: Use interruption mechanism or cooperative cancellation

**Thread.suspend() and Thread.resume():**

**------------------------------------------------------**

suspend(): Pauses thread execution. Prototype: **public void suspend()**

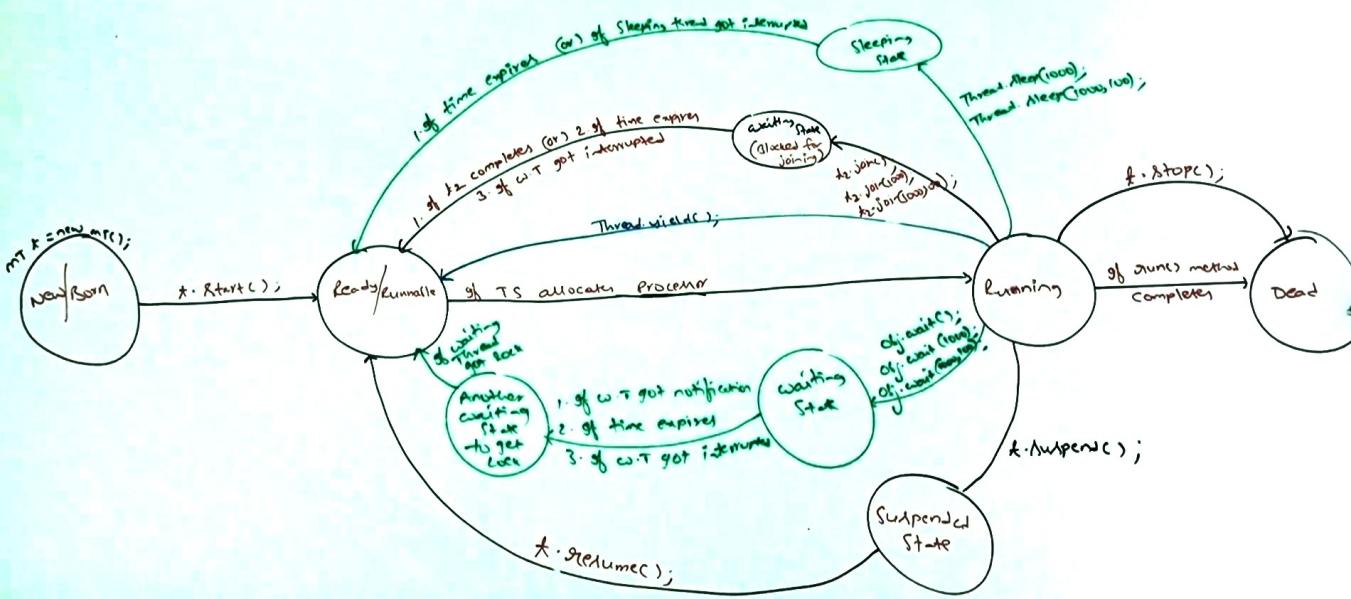
resume(): Continues execution of suspended thread. Prototype: **public void resume()**

(Both deprecated due to deadlock risks. suspend() holds locks while thread is suspended.)

Alternative: Use wait()/notify() pattern or java.util.concurrent tools. Modern thread management relies on interruption, ExecutorService, or other concurrent utilities from java.util.concurrent.

**Complete life cycle of thread:**

**----------------------------------------**

****

**Thread Group in Java:**

**------------------------------**

A ThreadGroup is a class in Java that represents a group of threads. It allows you to manage multiple threads as a single unit. Thread groups can contain not only threads but also other thread groups, forming a hierarchy.

Key Features of ThreadGroup:

----------------------------------------

**Hierarchy:** Thread groups can be nested, forming a tree-like structure. A thread group can contain threads and other thread groups.

**Security:** Thread groups can be used to restrict access to threads (e.g., prevent one thread group from modifying another).

**Exception Handling:** You can define a default uncaught exception handler for all threads in a group.

Advantages of ThreadGroup:

---------------------------------------

**Organized Management:** Thread groups allow you to manage multiple threads as a single unit, making it easier to control and monitor them.

**Bulk Operations:** You can perform operations (e.g., interrupt, set priority) on all threads in a group at once. Every Java

thread belongs to a thread group. The main thread belongs to **the "main" thread group.** Every thread group is a child of the

**system group.**

public class Test {

public static void main(String[] args) {

System.out.println(Thread.currentThread().getThreadGroup().getName()); // main

System.out.println(Thread.currentThread().getThreadGroup().getParent().getName()); // system

}

}

**Output:** main

system

**System Thread Group:** Root of all thread groups in Java. Contains important system-level threads like:

Finalizer (handles object finalization)

Reference Handler (manages weak references)

Attach Listener (used for debugging & monitoring)

Signal Dispatcher (handles OS signals)

Thread groups help manage multiple threads together, but they are rarely used in modern Java.

**→** ThreadGroup is indeed a Java class in the **java.lang package** that directly extends **Object**. It allows you to group

related threads together for collective management. ThreadGroup constructors:

**public ThreadGroup(String name):** Creates a thread group with the specified name. Parent is the thread group of the current thread.

**public ThreadGroup(ThreadGroup parent, String name):** Creates a thread group with the specified parent group and name.

Allows explicit specification of parent-child relationship. ( These constructors create a hierarchical structure of thread

groups for organized thread management.)

public class Test {

public static void main(String[] args) {

ThreadGroup g1 = new ThreadGroup("First");

System.out.println(g1.getParent().getName()); // main

ThreadGroup g2 = new ThreadGroup(g1, "Second");

System.out.println(g2.getParent().getName()); // First

}

}

**Output:** main

First

**→** Here are the 13 key methods of ThreadGroup:

**int activeCount() -** Returns estimate of **active threads** in group and subgroups.

**int activeGroupCount() -** Returns estimate of **active subgroups.**

**void destroy() -** Destroys empty group and subgroups.

**int enumerate(Thread[] list) -** Copies active threads in group and subgroups into array.

**int enumerate(ThreadGroup[] list) -** Copies active subgroups into array.

**String getName() -** Returns group name.

**int getMaxPriority() -** Returns maximum priority.

**void interrupt() -** Interrupts all threads in group.

**boolean isDaemon() -** Tests if group is daemon.

**void setDaemon(boolean daemon) -** Sets daemon status of group.

**void setMaxPriority(int pri) -** Sets maximum priority for this group and subgroups, doesn't affect **existing threads with higher priorities**, only newly created ones. Ex:

public class Test {

public static void main(String[] args) {

ThreadGroup g1 = new ThreadGroup("First");

Thread t1 = new Thread(g1, "Thread-1");

Thread t2 = new Thread(g1, "Thread-2");

g1.setMaxPriority(3);

Thread t3 = new Thread(g1, "Thread-3");

System.out.println(t1.getPriority());

System.out.println(t2.getPriority());

System.out.println(t3.getPriority());

}

}

**Output:** 5

5

3

**ThreadGroup getParent() -** Returns the parent thread group.

**void list() -** Prints information about the thread group to standard output, including group name, threads, and subgroups.

**Example-1:**

public class Test {

public static void main(String[] args) throws InterruptedException {

ThreadGroup pg = new ThreadGroup("Parent-group");

MyThread t1 = new MyThread(pg, "Child-1");

MyThread t2 = new MyThread(pg, "Child-2");

ThreadGroup cg = new ThreadGroup(pg, "Child-group");

t1.start();

t2.start();

System.out.println(pg.activeCount());

System.out.println(pg.activeGroupCount());

pg.list();

Thread.sleep(10000);

System.out.println(pg.activeCount());

System.out.println(pg.activeGroupCount());

pg.list();

}

}

class MyThread extends Thread {

MyThread(ThreadGroup g1, String name) {

super(g1, name);

}

@Override

public void run() {

System.out.println(this.getName()+" thread started");

try {

Thread.sleep(5000);

} catch (InterruptedException e) {

e.getMessage();

}

}

}

**Output** (Order May Vary)**:**

**-----------------------------------**

2

1

java.lang.ThreadGroup[name=Parent-group,maxpri=10]

Child-1 thread started

Child-2 thread started

Thread[#21,Child-1,5,Parent-group]

Thread[#22,Child-2,5,Parent-group]

java.lang.ThreadGroup[name=Child-group,maxpri=10]

0

1

java.lang.ThreadGroup[name=Parent-group,maxpri=10]

java.lang.ThreadGroup[name=Child-group,maxpri=10]

**Example-2:**

**---------------**

public class Test {

public static void main(String[] args) throws InterruptedException {

ThreadGroup system = Thread.currentThread().getThreadGroup().getParent();

Thread[] t = new Thread[system.activeCount()];

system.enumerate(t);

for (Thread t1 : t) {

System.out.println(t1.getName()+"...."+t1.isDaemon());

}

}

}

**Output:**

main....false

Reference Handler....true

Finalizer....true

Signal Dispatcher....true

Attach Listener....true

Notification Thread....true

Common-Cleaner....true

**Problems with traditional synchronized keyword:**

**-------------------------------------------------------------------**

**Performance overhead:** Lock acquisition/release requires expensive JVM operations. Contended locks cause thread context

switching. Synchronized blocks create memory barriers affecting CPU optimization.

**Limited control:** No timeout capability - threads block indefinitely. Can't abandon lock acquisition attempts after a certain period.

**Inflexible locking:** No tryLock() functionality to attempt lock without blocking. Cannot query lock status programmatically.

**No fairness policy:** Lock acquisition order isn't guaranteed. Possible thread starvation under high contention.

**All-or-nothing access:** No distinction between readers and writers. Multiple readers blocked unnecessarily.

**Nested synchronization risks:** Deadlock potential with nested or hierarchical locking. No deadlock detection capabilities.

**No interruption support:** Thread.interrupt() doesn't affect synchronized blocks. Can't cancel pending lock acquisition.

**Coarse-grained locking:** Either entire method or block must be synchronized. Difficulty implementing fine-grained concurrency.

Modern alternatives: java.util.concurrent.locks package, especially ReentrantLock and ReadWriteLock.

**java.util.concurrent.locks Package in Java:**

**---------------------------------------------------------**

The java.util.concurrent.locks package provides advanced locking mechanisms for more flexible and powerful thread synchronization compared to the traditional synchronized keyword. It includes interfaces and classes for explicit locks, conditions, and read-write locks. **Introduced in Java version 5.**

Key Features of java.util.concurrent.locks:

--------------------------------------------------------

**Explicit Locking:** Provides more control over locking and unlocking compared to synchronized.

**Condition Variables:** Allows threads to wait for specific conditions to be met.

**Read-Write Locks:** Supports separate locks for reading and writing, improving performance in read-heavy scenarios.

**Interruptible Locking:** Locks can be acquired in an interruptible manner.

**Timeout Support:** Locks can be acquired with a timeout, preventing deadlocks.

Main Classes and Interfaces:

**Class/Interface Description**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------**

**Lock** An interface for explicit locks. Replaces synchronized blocks/methods.

**ReentrantLock** A concrete implementation of Lock. Supports reentrant locking.

**ReadWriteLock** An interface for read-write locks.

**ReentrantReadWriteLock** A concrete implementation of ReadWriteLock.

**Condition**  Provides thread coordination similar to wait() and notify().

Advantages of java.util.concurrent.locks:

-------------------------------------------------------

**Flexibility:** Locks can be acquired and released in any order, unlike synchronized.

**Fine-Grained Control:** Supports multiple Condition objects per lock, allowing precise thread coordination.

**Debugging:** Provides better debugging support (e.g., identifying which thread holds a lock).

**Explanation of Lock Interface Methods:**

**------------------------------------------------------**

The Lock interface in the java.util.concurrent.locks package provides explicit locking mechanisms that are more flexible and powerful than the traditional synchronized keyword. Here’s a detailed explanation of the important methods of the Lock interface:

**void lock():**

**----------------**

Behavior:Acquires the lock if it is available. If the lock is not available, the thread waits until it can acquire the lock.

Similarity to synchronized: This method behaves exactly like the implicit lock acquired by a thread when entering a

synchronized block or method. **Example:**

Lock lock = new ReentrantLock();

lock.lock(); // Acquire the lock

try {

// Critical section

} finally {

lock.unlock(); // Release the lock

}

**boolean tryLock():**

**-------------------------**

Behavior: Attempts to acquire the lock without waiting. If the lock is available, the thread acquires it and returns true. If the lock is not available, the method returns false immediately, and the thread continues execution without waiting.

Use Case: Useful for avoiding deadlocks or when you want to perform an action only if the lock is available. Example:

Lock lock = new ReentrantLock();

if (lock.tryLock()) { // Try to acquire the lock

try {

// Critical section

} finally {

lock.unlock(); // Release the lock

}

} else {

// Perform alternative actions

}

**boolean tryLock(long time, TimeUnit unit):**

**----------------------------------------------------------**

Behavior: Attempts to acquire the lock within the specified time. If the lock is available, the thread acquires it and returns true. If the lock is not available, the thread waits for the specified time. If the lock is still not available after the timeout, the method returns false, and the thread continues execution.

Use Case: Useful for preventing indefinite waiting and avoiding deadlocks. Example:

Lock lock = new ReentrantLock();

try {

if (lock.tryLock(2, TimeUnit.SECONDS)) { // Wait for 2 seconds

try {

// Critical section

} finally {

lock.unlock(); // Release the lock

}

} else {

// Perform alternative actions

}

} catch (InterruptedException e) {

System.out.println("Thread interrupted while waiting for the lock.");

}

**What is TimeUnit?**

**-------------------------**

**→** The TimeUnit enum is part of the java.util.concurrent package in Java. TimeUnit is an **enumeration** that represents time units, such as DAYS, HOURS, MICROSECONDS, MILLISECONDS, MINUTES, NANOSECONDS, SECONDS. It provides a convenient way to specify time durations in a readable and type-safe manner.

**void lockInterruptibly():**

**--------------------------------**

Behavior: Acquires the lock if it is available and returns immediately. If the lock is not available, the thread waits until it can acquire the lock. While waiting, if the thread is interrupted, it throws an InterruptedException.

Use Case: Useful when you want to allow the thread to be interrupted while waiting for the lock. Example:

Lock lock = new ReentrantLock();

try {

lock.lockInterruptibly(); // Acquire the lock (interruptible)

try {

// Critical section

} finally {

lock.unlock(); // Release the lock

}

} catch (InterruptedException e) {

System.out.println("Thread interrupted while waiting for the lock.");

}

**void unlock():**

**-------------------**

Behavior: Releases the lock. The thread calling this method must be the owner of the lock (i.e., it must have acquired the lock). If the thread does not own the lock,it throws an IllegalMonitorStateException.

Use Case: Used to release the lock after completing the critical section. Example:

Lock lock = new ReentrantLock();

lock.lock(); // Acquire the lock

try {

// Critical section

} finally {

lock.unlock(); // Release the lock

}

**Key Points:**

**---------------**

**Explicit Locking:** The Lock interface provides more control over locking and unlocking compared to synchronized.

**Interruptibility:** Methods like lockInterruptibly() and tryLock() support interruptible locking.

**Timeout:** The tryLock(long time, TimeUnit unit) method supports acquiring locks with a timeout.

**ReentrantLock:** The most common implementation of the Lock interface is ReentrantLock.

**Unlocking:** Always release the lock in a finally block to ensure it is released even if an exception occurs.

**Comparison with synchronized:**

**Feature synchronized Lock Interface**

**--------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Lock Acquisition Implicit Explicit

Interruptibility Not interruptible Supports interruptible locking

Timeout No timeout support Supports timeout

Multiple Conditions Single condition (wait()/notify()) Supports multiple Condition objects

Reentrant Yes Yes (ReentrantLock)

**ReentrantLock Class in Java:**

**--------------------------------------**

The ReentrantLock class is a concrete implementation of the Lock interface in the java.util.concurrent.locks package. It provides reentrant mutual exclusion locks with the same basic behavior as the implicit locks used by synchronized methods and blocks but with additional features like interruptible locking, timeout support, and fairness policies.

Key Features of ReentrantLock:

------------------------------------------

**Reentrant:** A thread can acquire the same lock multiple times without deadlocking itself.

**Fairness:** Supports a fairness policy to ensure that the longest-waiting thread acquires the lock first.

**Condition Support:** Provides Condition objects for advanced thread coordination.

**→ ReentrantLock maintains a thread-specific count internally.** Each time a thread calls lock(), the count increments. Each time the thread calls unlock(), the count decrements. The lock is released only when the count reaches 0.

Constructors of ReentrantLock:

------------------------------------------

**ReentrantLock():** Creates a ReentrantLock with an unfair locking policy (default).

**ReentrantLock(boolean fair):** Creates a ReentrantLock with a specified fairness policy. If true, the lock uses a fair ordering policy (longest-waiting thread gets the lock first).

**Important Methods of ReentrantLock:**

**----------------------------------------------------**

**void lock():** Acquires the lock (waits if necessary).

**void lockInterruptibly():** Acquires the lock (interruptible).

**boolean tryLock():** Attempts to acquire the lock without waiting.

**void unlock():** Releases the lock.

**boolean isLocked():** Checks if the lock is held by any thread.

**boolean isHeldByCurrentThread():** Checks if the lock is held by the current thread.

**boolean isFair():** Checks if the lock uses a fair ordering policy.

**int getQueueLength():** Returns the approximate number of threads waiting for the lock.

**Collection getQueuedThreads():** Returns a collection of threads currently waiting for the lock.

**boolean hasQueuedThreads():** Returns true if any thread is waiting for the lock; otherwise, false.

**int getHoldCount():** Returns the number of times the current thread has acquired the lock.

**boolean tryLock(long time, TimeUnit unit):** Attempts to acquire the lock within the specified time.

**Thread getOwner():** Returns the thread that currently holds the lock, or null if the lock is not held.

Example:

-------------

import java.util.concurrent.locks.ReentrantLock;

public class Test {

public static void main(String[] args) {

ReentrantLock l = new ReentrantLock(false);

l.lock();

l.lock();

System.out.println(l.isLocked());

System.out.println(l.isHeldByCurrentThread());

System.out.println(l.getQueueLength());

l.unlock();

System.out.println(l.getHoldCount());

System.out.println(l.isLocked());

l.unlock();

System.out.println(l.isLocked());

System.out.println(l.isFair());

}

}

**Output:** true

true

0

1

true

false

false

**Example of lock() and unlock():**

**------------------------------------------**

import java.util.concurrent.locks.ReentrantLock;

public class Test {

public static void main(String[] args) {

Display d = new Display();

MyThread t1 = new MyThread(d, "Krishna");

MyThread t2 = new MyThread(d, "Ram");

t1.start();

t2.start();

}

}

class MyThread extends Thread {

Display d;

String name;

MyThread(Display d, String name) {

this.d = d;

this.name = name;

}

@Override

public void run() {

d.wish(name);

}

}

class Display {

ReentrantLock l = new ReentrantLock();

public void wish(String name) {

l.lock();

for (int i = 1; i <= 3; i++) {

System.out.print("Hello ");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.getMessage();

}

System.out.println(name);

}

l.unlock();

}

}

Output: Hello Krishna

Hello Krishna

Hello Krishna

Hello Ram

Hello Ram

Hello Ram

**Example of tryLock():**

**-----------------------------**

import java.util.concurrent.locks.ReentrantLock;

public class Test {

public static void main(String[] args) {

MyThread t1 = new MyThread("Thread-1");

MyThread t2 = new MyThread("Thread-2");

t1.start();

t2.start();

}

}

class MyThread extends Thread {

static ReentrantLock l = new ReentrantLock();

MyThread(String name) {

super(name);

}

@Override

public void run() {

if (l.tryLock()) {

System.out.println(Thread.currentThread().getName() + " got the lock !");

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.getMessage();

}

l.unlock();

}

else {

System.out.println(Thread.currentThread().getName() + " unable to get the lock");

}

}

}

**Output** (Order May Vary)**:** Thread-1 got the lock !

Thread-2 unable to get the lock !

**Example of tryLock(long time, TimeUnit unit):**

**---------------------------------------------------------------**

import java.util.concurrent.TimeUnit;

import java.util.concurrent.locks.ReentrantLock;

public class Test {

public static void main(String[] args) {

MyThread t1 = new MyThread("Thread-1");

MyThread t2 = new MyThread("Thread-2");

t1.start();

t2.start();

}

}

class MyThread extends Thread {

static ReentrantLock l = new ReentrantLock();

MyThread(String name) {

super(name);

}

@Override

public void run() {

do {

try {

if (l.tryLock(5000, TimeUnit.MILLISECONDS)) {

System.out.println(Thread.currentThread().getName() + " got the lock and performing safe operations !");

Thread.sleep(25000);

System.out.println(Thread.currentThread().getName() + " released the lock !");

l.unlock();

break;

}

else {

System.out.println(Thread.currentThread().getName() + " unable to get the lock and will try again !");

}

} catch (InterruptedException e) {

System.out.println(e);

}

} while (true);

}

}

**Output :** Thread-1 got the lock and performing safe operations !

Thread-2 unable to get the lock and will try again !

Thread-2 unable to get the lock and will try again !

Thread-2 unable to get the lock and will try again !

Thread-2 unable to get the lock and will try again !

Thread-1 released the lock !

Thread-2 got the lock and performing safe operations !

Thread-2 released the lock !

**→ Thread Pool (Executor Framework)** was introduced in Java 1.5. It helps manage multiple threads efficiently, preventing performance and memory issues caused by creating new threads for each task.

**Executors.newFixedThreadPool(int n)** creates a thread pool with a fixed number of threads (n). It reuses existing threads instead of creating new ones. Example: **ExecutorService service = Executors.newFixedThreadPool(3);**

**→** You can submit a Runnable or Callable job to the executor service using the submit() method: **service.submit(job);**

It executes the given task using one of the available threads in the pool. Shutting Down the Executor Service To stop the executor service gracefully: **service.shutdown();**

This allows previously submitted tasks to complete before terminating the service. After calling shutdown(), new tasks won't be accepted. If you need to immediately stop all tasks, you can use: **service.shutdownNow();** This attempts to stop all actively executing tasks immediately.

**Example:**

**------------**

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class Test {

public static void main(String[] args) {

MyThread[] jobs = { new MyThread("Job-1"), new MyThread("Job-2"), new MyThread("Job-3"), new MyThread("Job-4"), new MyThread("Job-5"), new MyThread("Job-6") };

ExecutorService service = Executors.newFixedThreadPool(3);

for (MyThread job : jobs) {

service.submit(job);

}

service.shutdown();

}

}

class MyThread implements Runnable {

String name;

MyThread(String name) {

this.name = name;

}

@Override

public void run() {

System.out.println(name + " job stated by " + Thread.currentThread().getName());

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.getMessage();

}

System.out.println(name + " job completed by " + Thread.currentThread().getName());

}

}

**Output** (Order May Vary)**:**

**-----------------------------------**

Job-3 job stated by pool-1-thread-3

Job-1 job stated by pool-1-thread-1

Job-2 job stated by pool-1-thread-2

Job-3 job completed by pool-1-thread-3

Job-1 job completed by pool-1-thread-1

Job-2 job completed by pool-1-thread-2

Job-4 job stated by pool-1-thread-1

Job-5 job stated by pool-1-thread-2

Job-6 job stated by pool-1-thread-3

Job-4 job completed by pool-1-thread-1

Job-5 job completed by pool-1-thread-2

Job-6 job completed by pool-1-thread-3

**→** Thread pools are widely used in web servers and application servers to efficiently handle multiple client requests without

creating a new thread for each request. This improves performance, reduces memory overhead, and prevents excessive

thread creation, which can lead to resource exhaustion.

**→** Unlike Runnable, which does not return a result, Callable<T> allows a thread to return a result after execution. When a Callable is submitted to an ExecutorService, it returns a **Future<T>** object, which can be used to retrieve the result using the **get()** method. This ensures that the thread execution result is accessible after completion. Example:

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class Test {

public static void main(String[] args) throws Exception {

MyCallable[] jobs = { new MyCallable(10), new MyCallable(20), new MyCallable(30),

new MyCallable(40), new MyCallable(50) };

ExecutorService service = Executors.newFixedThreadPool(3);

for (MyCallable job : jobs) {

Future<Integer> f = service.submit(job);

System.out.println(f.get());

}

service.shutdown();

}

}

class MyCallable implements Callable {

int num;

MyCallable(int num) {

this.num = num;

}

@Override

public Object call() {

System.out.print(Thread.currentThread().getName() + " is responsible to find sum of first " + num + " natural numbers: ");

int sum = 0;

for (int i = 0; i <= num; i++) {

sum += i;

}

return sum;

}

}

**Output:**

pool-1-thread-1 is responsible to find sum of first 10 natural numbers: 55

pool-1-thread-2 is responsible to find sum of first 20 natural numbers: 210

pool-1-thread-3 is responsible to find sum of first 30 natural numbers: 465

pool-1-thread-1 is responsible to find sum of first 40 natural numbers: 820

pool-1-thread-2 is responsible to find sum of first 50 natural numbers: 1275

**Detailed comparison between Runnable and Callable:**

**Feature Runnable Callable**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------**

**Definition** Represents a task that runs in a separate Represents a task that runs in a

thread but does not return a result. separate thread and returns a result.

**Method** public void run() public T call() throws Exception

**Return Type** void (does not return anything) Returns a generic type T (can return a value)

**Exception Handling** Cannot throw checked exceptions Can throw checked exceptions using throws Exception

**Package** java.lang java.util.concurrent

**Introduced In**  Java 1.0 Java 1.5 (as part of concurrency utilities)

**Usage** Used when a task does not require Used when a task needs to return

a return value or exception handling. a result and handle exceptions.

**Execution** Submitted to a Thread or ExecutorService. Submitted to an ExecutorService and returns a Future<T> object.

**ThreadLocal in Java:**

**----------------------------**

The ThreadLocal class in Java provides thread-local variables. **These variables are unique to each thread**, meaning that each thread has its own independently initialized copy of the variable. This is useful for maintaining per-thread state without requiring synchronization.

Key Features of ThreadLocal:

---------------------------------------

**Thread-Specific Storage:** Each thread has its own copy of the variable.

**No Synchronization Required:** Since each thread has its own copy, there is no need for synchronization.

**Memory Efficiency:** Reduces the need for creating multiple instances of a variable for each thread.

**Common Use Cases:** Storing user sessions, transaction contexts, or thread-specific configuration.

**How ThreadLocal Works:** When a thread accesses a ThreadLocal variable, it gets its own independent copy of the variable. Changes made by one thread are not visible to other threads.

**→** Once a thread enters the dead state, all its local variables become eligible for garbage collection, provided there are no external references to them. The statement:

ThreadLocal tl = new ThreadLocal(); // Creates a ThreadLocal variable that maintains a separate value for each thread that accesses it.

**Key Methods of ThreadLocal:**

**----------------------------------------**

**T get():** Returns the value of the thread-local variable for the current thread.

**void set(T value):** Sets the value of the thread-local variable for the current thread.

**void remove():** Removes the value of the thread-local variable for the current thread.

**→** The **initialValue()** method in ThreadLocal returns the initial value associated with the current thread. (Default behavior: Returns **null** if not overridden.)

Customization: We can override initialValue() to provide a specific initial value.

Re-initialization: If a value is removed using remove(), it will be re-initialized when accessed again by calling initialValue().

Example - 1:

-----------------

public class Test {

public static void main(String[] args) {

ThreadLocal<String> tl = new ThreadLocal<>();

System.out.println(tl.get());

tl.set("Ram");

System.out.println(tl.get());

tl.remove();

System.out.println(tl.get());

}

}

**Output:** null

Ram

null

Example - 2:

-----------------

public class Test {

public static void main(String[] args) {

ThreadLocal<Object> tl = new ThreadLocal<>() {

public Object initialValue() {

return "Krishna";

}

};

System.out.println(tl.get());

tl.set("Ram");

System.out.println(tl.get());

tl.remove();

System.out.println(tl.get());

}

}

**Output:** Krishna

Ram

Krishna

Example - 3:

-----------------

import java.util.concurrent.locks.ReentrantLock;

public class Test {

public static void main(String[] args) {

CustomerThread c1 = new CustomerThread("Customer Thread-1");

CustomerThread c2 = new CustomerThread("Customer Thread-2");

CustomerThread c3 = new CustomerThread("Customer Thread-3");

CustomerThread c4 = new CustomerThread("Customer Thread-4");

c1.start();

c2.start();

c3.start();

c4.start();

}

}

class CustomerThread extends Thread {

CustomerThread(String name) {

super(name);

}

private static ReentrantLock l = new ReentrantLock();

private static Integer custId = 0;

private static ThreadLocal<Integer> tl = new ThreadLocal<>() {

protected Integer initialValue() {

return ++custId;

}

};

@Override

public void run() {

l.lock();

System.out.println(Thread.currentThread().getName() + " executing with customer id : " + tl.get());

l.unlock();

}

}

**Output:** Customer Thread-1 executing with customer id : 1

Customer Thread-2 executing with customer id : 2

Customer Thread-3 executing with customer id : 3

Customer Thread-4 executing with customer id : 4

**→** By default, thread-local variables in a parent thread are not automatically inherited by child threads. Each thread has its own independent copy of thread-local variables. **InheritableThreadLocal** allows child threads to inherit values from the parent thread.

Default Behavior: Child thread gets the **same value as the parent** thread.

Customization: Override **childValue(Object parentValue)** to provide a different value for the child thread.

Constructor:  **InheritableThreadLocal tl = new InheritableThreadLocal<>();**

Methods: Inherits all methods from ThreadLocal and adds **childValue()**

Example:

public class Test {

public static void main(String[] args) {

ParentThread parent = new ParentThread();

parent.start();

}

}

class ParentThread extends Thread {

public static InheritableThreadLocal<Object> tl = new InheritableThreadLocal<>() {

public Object childValue(Object parent) {

return "Child";

}

};

@Override

public void run() {

tl.set("Parent");

System.out.println("Parent value: "+tl.get());

ChildThread child = new ChildThread();

child.start();

}

}

class ChildThread extends Thread {

@Override

public void run() {

System.out.println("Child value: "+ ParentThread.tl.get());

}

}

Output: Parent value: Parent

Child value: Child

**Inner Classes**

**===============================================================================**

**→ An inner class** is a class defined inside another class. It is also known as a nested class. Inner classes are used to

logically group classes and interfaces in a place where they will only be used in one place. **They can access all the**

**members (fields and methods) of the outer class, including private members.** Inner classes are part of the encapsulation concept, as they can be hidden from other classes in the same package.

Advantages of Inner Classes:

---------------------------------------

**Encapsulation:** Inner classes can access private members of the outer class, providing better encapsulation.

**Code Readability:** They help in grouping related classes together, making the code more readable and maintainable.

**Specialization:** Inner classes can be used to create specialized implementations that are only relevant to the outer class.

**→** Inner classes are used when an object of one class is tightly associated with another class. This helps in logically grouping classes and improving encapsulation. The best example of an inner interface is the **Map interface and its nested Entry interface** in Java. The Map.Entry interface represents a key-value pair inside a Map, and it cannot exist independently without a Map.

Types of Inner Classes:

-------------------------------

**Non-static Inner Class (Member Inner Class):** Defined at the member level of the outer class (not static). It cannot exist without an instance of the outer class. Example:

class Outer {

class Inner {

void display() {

System.out.println("Inside Inner Class");

}

}

}

**Static Nested Class:** Defined with the static keyword inside the outer class. It can be accessed without creating an instance of the outer class. Example:

class Outer {

static class Inner {

void display() {

System.out.println("Inside Static Nested Class");

}

}

}

**Local Inner Class:** Defined inside a block, such as a method or constructor. It is accessible only within the block where it is defined. Example:

class Outer {

void display() {

class LocalInner {

void show() {

System.out.println("Inside Local Inner Class");

}

}

LocalInner inner = new LocalInner();

inner.show();

}

}

**Anonymous Inner Class:** A class without a name, defined and instantiated at the same time. Often used for implementing interfaces or extending classes on the fly. Example:

interface Greeting {

void greet();

}

class Outer {

void display() {

Greeting greeting = new Greeting() {

public void greet() {

System.out.println("Hello from Anonymous Inner Class");

}

};

greeting.greet();

}

}

**→** Inner classes can be declared as private, protected, or public, just like regular class members. **Non-static inner classes cannot have static members** (except for static final variables). The .class file of an inner class is named as **OuterClass$InnerClass.class.**

**→** The relationship between an inner class and its outer class is a **has-a (composition) relationship**, not an is-a relationship. When a Java file containing normal inner classes is compiled, it generates multiple .class files: One for the outer class (Outer.class). One for each inner class (Outer$Inner.class). Each class can be executed separately if it has a main method. However, inner classes depend on the outer class, so they cannot be run independently unless they are **static nested classes.**

Case - 1: Accessing inner class code from static area of outer class:

-----------------------------------------------------------------------------------------

public class Test {

class Inner {

void m1() {

System.out.println("Inner Class m1 Method");

}

}

public static void main(String[] args) {

Test t = new Test();

Test.Inner inner = t.new Inner();

inner.m1(); // Inner Class m1 Method

// Test.Inner inner = new Test().new Inner();

// inner.m1(); // Inner Class m1 Method

// new Test().new Inner().m1(); // Inner Class m1 Method

}

}

**Output:** Inner Class m1 Method

Case - 2: Accessing inner class code from instance area of outer class:

--------------------------------------------------------------------------------------------

public class Test {

class Inner {

void m1() {

System.out.println("Inner Class m1 Method");

}

}

void m2() {

Inner i = new Inner();

i.m1();

}

public static void main(String[] args) {

Test t = new Test();

t.m2();

}

}

**Output:** Inner Class m1 Method

**→** Accessing an inner class from the static area of the outer class and accessing it from outside the outer class follows the same approach. Since a non-static inner class is associated with an instance of the outer class, we must first create an instance of the outer class to access the inner class.

**→** In a regular inner class, we can access both static and non-static members of the outer class directly.

**→** Inside the inner class, this refers to the current inner class instance, and if we want to refer to the outer class instance, we use OuterClassName.this. Example:

public class Test {

int x = 10;

class Inner {

int x = 100;

void m1() {

System.out.println("Inner Class m1 Method");

int x = 1000;

System.out.println(x);

System.out.println(Inner.this.x + "...both are same..." + this.x);

System.out.println(Test.this.x);

}

}

public static void main(String[] args) {

new Test().new Inner().m1();

}

}

**Output:** Inner Class m1 Method

1000

100...both are same...100

10

Applicable modifiers for **Inner classes:** public, abstract, final, strictfp, default, static, private, protected

**→ Nesting of inner classes is perfectly valid** in Java. You can declare an inner class within another inner class, creating a hierarchy of nested classes. Example:

public class Test {

class Inner {

class InnerMost {

void m1() {

System.out.println("Test.Inner.InnerMost.m1()");

}

}

}

public static void main(String[] args) {

Test t = new Test();

Test.Inner inner = t.new Inner();

Test.Inner.InnerMost i = inner.new InnerMost();

i.m1();

}

}

**Output:** Test.Inner.InnerMost.m1()

**→** Method-level inner classes (also known as local inner classes) are primarily used for method-specific tasks that are repeatedly required. They are suitable for nested method requirements but are rarely used due to their limited scope.

Key Points: Defined inside a method and can only be used within that method. **Can access final or effectively final variables**

**of the method.** Cannot have static members because they are tied to an instance. Best used when a class is required only

within a specific method. Example:

public class Test {

void method() {

class Inner {

void sum(int n1, int n2) {

System.out.println("Sum of "+n1+" and "+n2+" = "+(n1+n2));

}

}

Inner i = new Inner();

i.sum(10, 20);

i.sum(100,200);

}

public static void main(String[] args) {

new Test().method();

}

}

Output: Sum of 10 and 20 = 30

Sum of 100 and 200 = 300

**→** Method-local inner classes can be declared inside both instance methods and static methods, but their behavior differs: **Inside an Instance Method:** Can access both static and non-static members of the outer class. Since the instance method is tied to an object, it allows access to non-static members.

**Inside a Static Method:** Can only access static members of the outer class. Cannot access non-static members because static methods belong to the class, not an instance. Example:

public class Test {

int x = 10;

static int y = 20;

static void method() {

class Inner {

void m1() {

System.out.println(x+"......"+y); // Compile-time error

}

}

Inner i = new Inner();

i.m1();

}

public static void main(String[] args) {

Test.method();

}

}

Error: non-static variable x cannot be referenced from a static context

**Why Can't We Declare Static Members Inside Inner Classes?** : Static members belong to the class itself, not to any instance of the class. Inner classes, being non-static, are tied to an instance of the outer class. Declaring a static member inside an inner class would create a conflict, as the static member would not be associated with any instance of the outer class.

Applicable modifiers for Method-Local Inner Classes: **final, abstract, strictfp**

Case - 1:

public class Test {

void method() {

int x = 100;

class Inner {

void m1() {

System.out.println(x);

}

}

Inner i = new Inner();

i.m1();

}

public static void main(String[] args) {

new Test().method();

}

} // **Output:** 100

**Explanation:** **Since Java 8,** local variables accessed from within a method-local inner class need not be explicitly declared as final as long as they are effectively final. In this code, x is assigned a value of 100 and is never modified, so it is effectively final. Therefore, the **inner class can access x without any compilation error.** Effectively final means a local variable or parameter is not **explicitly declared as final but is never modified** **after its initial assignment.**

**→** Anonymous inner classes are mainly used for **one-time use.** They provide a quick way to create a subclass or implement an interface without explicitly defining a separate class.

Types of Anonymous Inner Classes:

-----------------------------------------------

**Anonymous Inner Class that Extends a Class:** Used when we need to extend a class but don’t want to create a named subclass. Example:

public class Test {

public static void main(String[] args) {

Popcorn p = new Popcorn() {

void taste() {

System.out.println("Spicy");

}

};

p.taste();

Popcorn p1 = new Popcorn();

p1.taste();

Popcorn p2 = new Popcorn() {

public void taste() {

System.out.println("Sweet");

}

};

p2.taste();

System.out.println(p.getClass().getName());

System.out.println(p1.getClass().getName());

System.out.println(p2.getClass().getName());

}

}

class Popcorn {

void taste() {

System.out.println("Salty");

}

}

**Output:** Spicy

Salty

Sweet

Test$1

Popcorn

Test$2

Here generated **.class files** are: **Popcorn.class, Test.class, Test$1.class, Test$2.class**

**→** When we create an anonymous inner class, we are implicitly declaring a child class that extends a given class or implements an interface without a name. **We then create an instance of that unnamed child class using a parent class reference.**

Breakdown of Code:

Popcorn p = new Popcorn() { // Anonymous inner class extending Popcorn

void taste() {

System.out.println("Spicy");

}

};

p.taste(); // Output: Spicy

**Popcorn p →** p is a reference of the Popcorn parent class.

**new Popcorn() { ... } →** We are creating an anonymous subclass of Popcorn that overrides the taste() method.

**Overriding the taste() method →** The anonymous inner class provides a custom implementation of taste().

**Parent reference, child object →** We are assigning the child object (anonymous class) to the Popcorn reference.

**Important Points: It cannot define constructors since it doesn't have a name.** It is instantiated immediately where it is declared. Even though p is a Popcorn reference, at runtime, the overridden taste() method of the anonymous inner class is executed instead of Popcorn's original method.

This technique is widely used in **event handling and UI programming,** where we need short-lived subclass implementations.

Example - 2:

-----------------

public class Test {

public static void main(String[] args) {

Thread t = new Thread() {

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println("Child Thread - "+ i);

}

}

};

t.start();

for (int i = 1; i <= 3; i++) {

System.out.println("Main Thread - "+ i);

}

}

}

**Output (**May vary**):** Child Thread - 1

Child Thread - 2

Child Thread - 3

Main Thread - 1

Main Thread - 2

Main Thread - 3

**→ Anonymous Inner Class that Implements an Interface** Used when we need to provide an implementation for an interface. Example:

public class Test {

public static void main(String[] args) {

Runnable r = new Runnable() {

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println("Child Thread - "+ i);

}

}

};

Thread t = new Thread(r);

t.start();

for (int i = 1; i <= 3; i++) {

System.out.println("Main Thread - "+ i);

}

}

}

**Output (**May vary**):** Child Thread - 1

Child Thread - 2

Child Thread - 3

Main Thread - 1

Main Thread - 2

Main Thread - 3

**→ Anonymous Inner Class Inside Argument (Argument-Based)** Used when we pass an anonymous class as an argument to a method. Example:

public class Test {

public static void main(String[] args) {

new Thread(new Runnable() {

public void run() {

for (int i = 1; i <= 3; i++) {

System.out.println(Thread.currentThread().getName()+" : "+i);

}

}

}).start();

for (int i = 1; i <= 3; i++) {

System.out.println(Thread.currentThread().getName()+" : "+i);

}

}

}

**Output** (May vary): Thread-0 : 1

Thread-0 : 2

main : 1

Thread-0 : 3

main : 2

main : 3

**Comparison of Anonymous Inner Class vs Normal Class:**

**Feature Normal Class Anonymous Inner Class**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------------**

Extends & Implements Can extend a class and implement Can either extend a class or multiple interfaces implement an interface at a time, not both

Constructors Can have multiple explicit constructors Cannot have explicit constructors (because it has no name)

Reusability Used for standard, reusable code structures Used for one-time, instant use (temporary requirements)

Name Has a defined name Has no name (declared anonymously)

File Structure Can be written as a separate .java file Defined inside a method or block

Usage Suitable for large, complex logic requiring Best for short-lived objects, such as

multiple methods event handling, thread execution, etc.

**Difference Between Normal Inner Class and Static Nested Class:**

**Feature Normal (Non-static) Inner Class Static Nested Class**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------**

Association with Outer Class Strongly associated (Cannot exist Not strongly associated (Can

without an instance of outer class) exist independently)

Object Creation Requires an outer class object to Can be created without an outer

create its instance class instance

Access to Outer Class Members Can access both static and non-static Can **only access static members** of

members of the outer class the outer class

Declaration Declared without static keyword Declared with static keyword inside an

inside an outer class outer class

Use Case When the inner class needs to When the nested class can work independently

use outer class members without an instance of the outer class

Example:

public class Test {

static class Inner {

void m1() {

System.out.println("Test.Inner.m1()");

}

}

public static void main(String[] args) {

Inner i = new Inner();

i.m1();

}

}

**Output:** Test.Inner.m1()

**→** If you want to create nested class object outside of the outer class:  **Test.Inner i = new Test.Inner();**

**→ Because static nested classes can have static members, they can have a main method.** This allows them to be executed directly from the command line, just like top-level classes. Example:

public class Test {

static class Inner {

public static void main(String[] args) {

System.out.println("Test.Inner.main()");

}

}

public static void main(String[] args) {

System.out.println("Test.main()");

}

}

**Output:** > java Test$Inner

**Test.Inner.main()**

**Various combinations of nested classes and interfaces in Java:**

**------------------------------------------------------------------------------------**

**Class inside a Class (Inner Class):** A class can be defined inside another class. This is the most common type of nested class. The inner class can access all members of the outer class, including private members. An instance of the inner class cannot exist without an instance of the outer class.

**Interface inside a Class:** When multiple implementations of an interface are needed, and all these implementations are tightly related to a particular class, defining the interface inside that class helps in better encapsulation and code organization. This approach is useful in nested behaviors where the outer class logically groups related interface implementations. Example: Interface Inside a Class

class Computer {

// Nested interface

interface USB {

void connect();

}

// First implementation of USB interface

class Keyboard implements USB {

public void connect() {

System.out.println("Keyboard connected via USB");

}

}

// Second implementation of USB interface

class Mouse implements USB {

public void connect() {

System.out.println("Mouse connected via USB");

}

}

} ( The interface can be implemented by any class, including the outer class or other classes. The interface can be accessed using **Computer.USB**.)

**Interface inside an Interface:** An interface can be defined inside another interface. If an interface logically belongs to another interface, defining it inside the outer interface improves encapsulation and readability. A real-world example is **Map and Map.Entry in Java.**

A Map represents a collection of key-value pairs, and each key-value pair is represented as an Entry. Without a Map, there's no meaning to an Entry, so Entry is declared inside Map as a nested interface. This concept is useful when an interface is tightly bound to another interface, just like how Entry is tightly linked with Map.

**Key Points About Interfaces Inside Interfaces:**

**--------------------------------------------------------------**

**Always public and static by defaul**t **→** Whether we declare it explicitly or not.

Inner interfaces can be implemented **independently** **→** Implementing the outer interface does not require implementing the inner interface, and vice versa.

**→** Inner interface can be accessed using **OuterInterface.InnerInterface syntax.** Example:

public class Test implements Outer {

public void m1() {

System.out.println("Test.m1()");

}

public static void main(String[] args) {

new Test().m1();

new Test2().m2();

}

}

interface Outer {

void m1();

interface Inner {

void m2();

}

}

class Test2 implements Outer.Inner {

public void m2() {

System.out.println("Test2.m2()");

}

}

**Output:** Test.m1()

Test2.m2()

**Class inside an Interface:** A class can be defined inside an interface. This is useful when the class is closely related to the interface. Key Points About Declaring a Class Inside an Interface:

**Closely Associated Functionality** **→** If a class is highly related to an interface, it's recommended to define it inside the interface.

**Providing Default Implementation →** A class inside an interface can provide a default implementation for the interface.

**Always public and static by default** **→** Any class inside an interface is static whether explicitly declared or not.

**Example : Default Implementation Inside an Interface**

interface Vehicle {

void start();

class DefaultCar implements Vehicle {

public void start() {

System.out.println("Car is starting...");

}

}

}

public class Main {

public static void main(String[] args) {

Vehicle.DefaultCar car = new Vehicle.DefaultCar();

car.start(); // Output: Car is starting...

}

}

**Explanation:** DefaultCar is a static nested class inside the Vehicle interface. It provides a default implementation of the start() method. We can create an object of Vehicle.DefaultCar without implementing Vehicle separately.

( **Conclusion:** It can provide a default implementation or utility methods. Since classes inside interfaces are static, they can be accessed without an outer class instance.)

**→ An interface declared inside a class is implicitly static.** This means it behaves like a static nested class. While it's implicitly static, an interface declared inside a class does not have to be public. It can have any access modifier allowed for nested members, including: **public, protected, (default/package-private), private**

**Why implicitly static? :** An interface declares a set of methods that implementing classes must adhere to. It doesn't make sense for an interface to be associated with a specific instance of the outer class. Therefore, it's treated as a static member of the outer class.

**Java.lang package**

**===============================================================================**

**java.lang Package:** The java.lang package contains the most commonly used classes and interfaces in Java.

This package is **automatically imported** in every Java program, so we don’t need to import it explicitly.

Object Class: The Object class is the **parent class of all Java classes.** Every class in Java, whether predefined or custom-

defined, extends Object either directly or indirectly. The Object class provides some fundamental methods, such as:

toString(), equals(), hashCode(), clone(), finalize(), getClass(), wait(), notify(), notifyAll()

Because of this inheritance, every Java class automatically gets these methods. **Why Object is Called the Root of All Classes? :** Since every Java class extends Object, it forms the base of the entire Java class hierarchy. This makes it possible for Java to handle objects in a generic way, such as in collections (ArrayList<Object>), reflection, and serialization. Example: Object Class Methods in Action

class Demo {}

public class Main {

public static void main(String[] args) {

Demo d = new Demo();

System.out.println(d.toString()); // Output: Demo@HexHashCode

System.out.println(d.hashCode()); // Output: Some integer hashcode

System.out.println(d.getClass()); // Output: class Demo

}

}

**→** Here are the prototypes of all **11 methods** in the java.lang.Object class:

1. public final native Class<?> getClass()
2. public native int hashCode()
3. public boolean equals(Object obj)
4. protected native Object clone() throws CloneNotSupportedException
5. public String toString()
6. public final native void notify()
7. public final native void notifyAll()
8. public final void wait() throws InterruptedException
9. public final native void wait(long timeout) throws InterruptedException
10. public final void wait(long timeout, int nanos) throws InterruptedException
11. protected void finalize() throws Throwable

**→** The Object class in Java contains 12 methods, but only 11 are available to subclasses because the registerNatives() method is: **private static native void registerNatives();** This method is used internally by the JVM to register native methods for the Object class. Since it is private, it is not inherited by child classes, and we don’t need to consider it in regular Java development.

**→ The toString()** method is used to get the string representation of an object. Whenever we try to print an object reference, internally the toString() method is called. If a class does not override toString(), the default implementation from Object class is executed, which returns:

**getClass().getName() + "@" + Integer.toHexString(hashCode()).** Example:

public class Test {

public static void main(String[] args) throws Exception {

Student s1 = new Student(1, "Devi");

Student s2 = new Student(2, "Prasad");

System.out.println(s1);

System.out.println(s2.toString());

}

}

class Student {

int id;

String name;

Student(int id, String name) {

this.id = id;

this.name = name;

}

}

**Output:** Student@28a418fc

Student@5305068a

**Best practice:** **Always override toString()** for meaningful object representation! Example:

class Student {

String name;

int age;

Student(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Student(name = '" + name + "', age = " + age + ")";

}

}

public class Main {

public static void main(String[] args) {

Student s = new Student("John", 22);

System.out.println(s);

}

}

**Output** (Custom toString()): Student(name = 'John', age = 22)

**→ All wrapper classes** (Integer, Double, etc.), **collection classes** (ArrayList, HashMap, etc.), and **string-related classes** (String, StringBuffer, StringBuilder) override the toString() method to provide a meaningful string representation.

**Why Should We Override toString()?**

**--------------------------------------------------**

For better debugging: Helps in logging meaningful object details.

For better readability: Instead of memory addresses, we get useful information.

To match Java standards: As Java core classes override it, our classes should too.

**→** For every object, a unique number is generated by the JVM, **which is its hash code.** The hash code does not represent the object's memory address. The JVM uses hash codes when storing objects in hashing-related data structures, such as hash tables, HashMaps, and HashSets. The main advantage of storing objects based on their hash code is that search operations become **more efficient**. Hashing is a highly efficient search algorithm.

**→** If the Object class's hashCode() method is used, it generates a hash code based on the object's memory address. This does not mean the hash code represents the object's memory address. Based on our requirements, we can override the hashCode() method in our class to generate our own hash code.

**→** Overriding the hashCode() method is considered proper if and only if it generates a unique hash code for each distinct object. For example:

class Student {

int rollNo;

Student(int rollNo) {

this.rollNo = rollNo;

}

@Override

public int hashCode() {

return rollNo;

}

}

**Handle potential collisions:** Simple implementations can lead to many objects having the same hashCode, which reduces

the performance of hash-based collections.

**Follow the contract with equals():** If you override equals(), you must also override hashCode(), and they must be consistent. If equals() returns true for two objects, their hashCode() must return the same value.

**→** If the Object class's toString() method is used, it internally incorporates the object's hash code. If we override the toString() method, our overridden toString() method may or may not call the hashCode() method, depending on our implementation. **Example:**

public class Test {

int i;

Test(int i) {

this.i = i;

}

@Override

public int hashCode() {

return i;

}

public static void main(String[] args) throws Exception {

Test t1 = new Test(10);

Test t2 = new Test(100);

System.out.println(t1);

System.out.println(t2);

}

}

**Output:**  Test@a

Test@64

**Explanation:** In this example, only the hashCode() method is overridden. Therefore, when System.out.println(t1) and System.out.println(t2) are executed, the Object class's toString() method is invoked. This results in output in the format: classname@hexadecimal representation of the object's hash code. In this case, since hashCode() returns the integer i, **the hexadecimal representation of i will be used.**

**→ We can use the equals() method to check the equality of two objects.** If our class does not contain an overridden equals() method, then the Object class's equals() method will be executed. Example:

public class Test {

public static void main(String[] args) throws Exception {

Student s1 = new Student(1, "Devi");

Student s2 = new Student(1, "Devi");

Student s3 = s1;

System.out.println(s1.equals(s2));

System.out.println(s1.equals(s3));

}

}

class Student {

int rollNo;

String name;

Student(int rollNo, String name) {

this.rollNo = rollNo;

this.name = name;

}

}

**Output:** false

true

**Explanation:** Since the Student class does not override the equals() method, the default implementation from Object class is used. The Object class equals() method performs reference comparison, meaning it checks if both references point to the same object in memory. **If content-based comparison is required, always override equals().**

**→** When overriding the equals() method for content comparison, we must consider the following:

Define the meaning of equality based on the specific requirements of the class. Ensure that our equals() method does not throw a **ClassCastException when passed an object of a different type.** Handle potential ClassCastException exceptions by returning false. Ensure that our equals() method does not throw a **NullPointerException when passed a null argument.** Handle potential NullPointerException exceptions by returning false. Example:

class Student {

int rollNo;

String name;

Student(int rollNo, String name) {

this.rollNo = rollNo;

this.name = name;

}

@Override

public boolean equals(Object obj) {

try {

Student s1 = (Student) obj;

if (name.equals(s1.name) && rollNo == s1.rollNo) {

return true;

}

else {

return false;

}

} catch (ClassCastException | NullPointerException e) {

return false;

}

}

}

After overriding equals():

---------------------------------

public static void main(String[] args) {

Student s1 = new Student(1, "Devi");

Student s2 = new Student(1, "Devi");

Student s3 = s1;

System.out.println(s1.equals(s2));

System.out.println(s1.equals(s3));

}

}

**Output:** true

true

**More simplified version of equals():**

**-------------------------------------------------**

class Student {

int rollNo;

String name;

Student(int rollNo, String name) {

this.rollNo = rollNo;

this.name = name;

}

@Override

public boolean equals(Object obj) {

if (this == obj) {

return true;

}

if (obj instanceof Student) {

Student s1 = (Student) obj;

if (name.equals(s1.name) && rollNo == s1.rollNo) {

return true;

} else {

return false;

}

}

else {

return false;

}

}

} // In the provided equals() method, if both references point to the same object (i.e., this == obj), the equals() method returns true immediately, without performing any further comparisons.

**→ In the String class, the equals() method is overridden for content comparison.** Therefore, even if two String objects are distinct instances, if their character sequences are identical, the equals() method returns true. However, in **the StringBuffer class, the equals() method is not overridden for content comparison.** Therefore, if two distinct StringBuffer objects have the same character sequence, the equals() method will still return false because it uses the default Object class's equals() method, which performs reference equality.

**→** We can use the **getClass() method** to obtain the runtime class definition of an object. Its prototype is **public final Class<?> getClass();** By using this Class object, we can access class-level properties such as the fully qualified name of the class, method information, constructor information, and other metadata. Example:

import java.lang.reflect.\*;

public class Test {

public static void main(String[] args) {

Object obj = new String("Devi");

Class c = obj.getClass();

Method[] methods = c.getDeclaredMethods();

int count = 0;

for (Method m : methods) {

count++;

}

System.out.println("Number of methods in "+c.getName()+" class = "+count);

}

}

**Output:** Number of methods in java.lang.String class = 148

**→** After loading each .class file, the JVM creates an object of type **java.lang.Class in the heap area.** Programmers can use this Class object to retrieve class-level information. The getClass() method is frequently used in reflection. Ex:

Connection con = DriverManager.getConnection(url, user, password);

// Get the class information of the Connection object

Class<?> connectionClass = con.getClass();

// Print the fully qualified class name

System.out.println("Connection class: " + connectionClass.getName());

**DriverManager.getConnection():** This method returns a Connection object, but the actual class of the Connection object is determined by the specific JDBC driver you are using. It is a vendor specific implementation of the Connection interface.

**con.getClass():** This line retrieves the runtime class of the Connection object.

**connectionClass.getName():** This gets the fully qualified name of the class (e.g., com.mysql.cj.jdbc.ConnectionImpl).

**→** Just before destroying an object, the **garbage collector calls the finalize() method** to perform cleanup activities. Once the finalize() method completes, the garbage collector automatically destroys the object.

**We can use wait(), notify(), and notifyAll() for inter-thread communication.** The thread that expects an update is responsible for calling the wait() method, which immediately causes the thread to enter a waiting state. The thread responsible for performing the update, after completing the update, can call the notify() method. The waiting thread will receive the notification and continue its execution with the updated data.

**String class:**

**----------------**

public class Test {

public static void main(String[] args) {

String str = "Devi";

StringBuffer sb = new StringBuffer("Devi");

str.concat("Prasad");

sb.append("Prasad");

System.out.println(str);

System.out.println(sb);

}

}

**Output:** Devi

DeviPrasad

**→** Once a String object is created, its value cannot be changed. **If any modification is attempted, a new String object is created with the updated value.** This non-changeable behavior is known as the **immutability of String objects.**

**Once a StringBuffer object is created, its value can be changed.** This changeable behavior is known as the mutability of StringBuffer objects.

Case - 1:

------------

String str1 = new String("Devi");

String str2 = new String("Devi");

StringBuffer sb1 = new StringBuffer("Devi");

StringBuffer sb2 = new StringBuffer("Devi");

System.out.println(str1 == str2);

System.out.println(str1.equals(str2));

System.out.println(sb1 == sb2);

System.out.println(sb1.equals(sb2));

**Output:**  false

true

false

false

**Explanation:** In the String class, the equals() method is overridden for content comparison. However, **in the StringBuffer class, the equals() method is not overridden** for content comparison.

**→ String s = new String("Java") :** In this case, two objects may be created. One object is created in the heap area, and

another object may be created in the String Constant Pool (SCP). s always points to the object in the heap.

**String s = "Java" :** In this case, one object is created in the SCP, and s always points to that object.

**Object creation in the SCP is conditional.** The JVM first checks if an object with the required content already exists in the SCP. If an object with the same content is found, the existing object is reused. If such an object is not found, a new object is created in the SCP but this rule is applicable only for SCP but not for Heap.

**( The garbage collector does not directly access the String Constant Pool (SCP).** Even if a String object in the SCP has no active references, it is not eligible for garbage collection. Prior to Java 7 (1.7), the String Constant Pool (SCP) was located in the **method area** (or PermGen). Starting with Java 7, the SCP was moved to the heap area. This change was implemented to improve memory utilization and reduce the risk of OutOfMemoryError exceptions related to the PermGen space. (The **garbage collector primarily operates on the heap area.)**

PermGen, short for Permanent Generation, was a memory space in the older versions of the Java Virtual Machine (JVM), specifically **before Java 8.** It was part of the **heap** and was used to store **metadata** like class definitions, method definitions, static variables, and string interned objects. In Java 8, PermGen was removed and replaced with Metaspace. Metaspace is allocated from **native memory**, rather than the Java heap, which eliminates the fixed-size limitation and reduces the risk of OutOfMemoryError exceptions.**)**

**→** All objects in the String Constant Pool (SCP) are typically released when the **JVM shuts down.** Whenever the new operator is used, a new object is created in the heap area. Therefore, it is possible to have multiple objects with the same content in the heap area, but not in the SCP. Duplicate string objects can exist in the heap area, but not in the SCP.

For every string literal, an object is placed in the SCP. **If an object is created during runtime (e.g., through string manipulation or concatenation), that object will be placed in the heap area, not the SCP.**

**Here are the String constructors:**

**---------------------------------------------**

**String():** Constructs an empty string.

**String(String original):** Constructs a new String that is a copy of the given String.

**String(StringBuffer buffer):** Constructs a new String from a StringBuffer.

**String(StringBuilder sb):** Constructs a new String from a StringBuilder.

**String(char[] value):** Constructs a new String from a character array.

**String(byte[] bytes):** Constructs a new String by decoding the specified array of bytes using the platform's

default charset.

**Important Methods of the String Class:**

**-----------------------------------------------------**

**public char charAt(int index):** Returns the character at the specified index.

**public String concat(String str):** Appends the specified string to the end of this string.

**public boolean equals(Object o):** Compares this string to the specified object for equality (content comparison).

**public boolean equalsIgnoreCase(String s):** Compares this string to the specified string, ignoring case considerations.

**public String substring(int begin):**  Returns a substring starting from the specified begin index to the end of the string.

**public String substring(int begin, int end):** Returns a substring from the specified begin index to the specified end index (exclusive).

**public int length():** Returns the length of this string.

**public String replace(char oldchar, char newchar):**  Returns a new string resulting from replacing all occurrences of oldchar in this string with newchar.

**public String toLowerCase():** Returns a new string with all characters converted to lowercase.

**public String toUpperCase():** Returns a new string with all characters converted to uppercase.

**public String trim():** Returns a new string with leading and trailing whitespace removed.

**public int indexOf(int ch):** Returns the index within this string of the first occurrence of the specified character, or -1 if the

character does not occur.

**public int indexOf(String str):** Returns the index within this string of the first occurrence of the specified substring.

**public int lastIndexOf(int ch):** Returns the index within this string of the last occurrence of the specified character, or -1 if the character does not occur.

**public int lastIndexOf(String str):** Returns the index within this string of the last occurrence of the specified substring.

**public boolean isEmpty():** Returns true if the string has a length of 0, and false otherwise.

Case - 2:

------------

String s1 = new String("devi");

String s2 = s1.toLowerCase();

String s3 = s1.toUpperCase();

String s4 = s3.toLowerCase();

System.out.println(s1 == s2);

System.out.println(s1 == s3);

System.out.println(s1 == s4);

**Output:** true

false

false

**→** If a runtime operation results in a change to the content of a String, a new String object will be created in the heap. **If there is no change to the content, the existing object will be used, and a new object will not be created.** This rule applies whether the object is located in the heap or the String Constant Pool (SCP).

**→** **Once an immutable object is created, its state cannot be changed.** If a modification is attempted, and the modification results in a change to the object's state, a new object with the updated state is created. If the attempted modification does not change the object's state, the existing object is reused. This behavior is known as **immutability.** We can create our own immutable classes. Here's a complete example:

public final class Test { // Make class final to prevent subclassing

private final int i; // Make field final and private

public Test(int i) {

this.i = i;

}

public int getI() { // Provide getter method, no setter

return i;

}

public Test modify(int i) {

if (this.i == i) {

return this; // Return existing object if no change

} else {

return new Test(i); // Create new object with modified state

}

}

public static void main(String[] args) {

Test t1 = new Test(10);

Test t2 = t1.modify(10);

Test t3 = t1.modify(20);

System.out.println(t1 == t2); // Output: true

System.out.println(t1 == t3); // Output: false

}

}

Key Elements for Immutability:

------------------------------------------

**final Class:** Declare the class as final to prevent subclassing, which could potentially violate immutability.

**final and private Fields:** Declare all instance variables as final and private. This ensures that they can only be initialized once (in the constructor) and cannot be modified later.

**→ final is applicable to variables, but not directly to objects.** Immutability is applicable to objects, but not directly to variables. Declaring a reference variable as final does not make the object it refers to immutable. Even though the reference variable is final, we can still modify the state of the corresponding object. However, we cannot reassign the final variable to refer to a different object. **(Therefore, final and immutability are distinct concepts.) Example:**

final StringBuffer sb = new StringBuffer("Devi");

sb.append("Prasad");

System.out.println(sb);

// sb = new StringBuffer("Dash"); error: cannot assign a value to final variable sb

**Output:** DeviPrasad

**StringBuffer in Java:**

**----------------------------**

**Definition:** StringBuffer is a **mutable sequence of characters.** It is part of the java.lang package. Unlike String, which is immutable, StringBuffer allows modifications to its content **without creating a new object.**

**Purpose:** It is used when frequent modifications to strings are required (e.g., concatenation, insertion, deletion). It is more efficient than String for such operations because it avoids creating multiple objects in memory.

Key Features:

------------------

**Mutable:** The content of a StringBuffer can be changed after creation.

**Thread-Safe:** All methods in StringBuffer are synchronized, making it safe for use in multi-threaded environments.

**Performance:** Slower than StringBuilder due to synchronization overhead but safer for multi-threading.

**Constructors of StringBuffer:**

**---------------------------------------**

StringBuffer(): Creates an empty StringBuffer with an initial capacity of **16 characters.**

Example: **StringBuffer sb = new StringBuffer();**

**→** When a StringBuffer reaches its capacity, a new character array is created. The new character array will have a new capacity calculated as **(current capacity \* 2) + 2.** The contents of the old character array are copied into the new array, and the StringBuffer object's internal reference is updated to point to the new array. The StringBuffer object itself remains the same.

StringBuffer(int capacity): Creates an empty StringBuffer with the specified initial capacity. Example:

**StringBuffer sb = new StringBuffer(100);**

StringBuffer(String str): Creates a StringBuffer with the specified string and an **initial capacity of 16 + length of the string.**

Example: **StringBuffer sb = new StringBuffer("Hello");**

**Common Methods of StringBuffer:**

**-----------------------------------------------**

**append():** Adds the specified data (e.g., string, int, char, etc.) to the end of the StringBuffer. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // "Hello World"

**insert():** Inserts the specified data at the given position. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.insert(1, "Java"); // "HJavaello"

**delete():** Removes characters from the specified start index to the end index. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.delete(1, 3); // "Hlo"

**deleteCharAt():** Removes the character at the specified index. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.deleteCharAt(1); // "Hllo"

**reverse():** Reverses the characters in the StringBuffer. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.reverse(); // "olleH"

**replace():** Replaces the characters from the specified start index to the end index with the given string. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.replace(1, 3, "Java"); // "HJavalo"

**capacity():** Returns the current capacity of the StringBuffer. Example:

StringBuffer sb = new StringBuffer();

System.out.println(sb.capacity()); // 16

**ensureCapacity():** Ensures that the capacity is at least equal to the specified minimum. Example:

StringBuffer sb = new StringBuffer();

sb.ensureCapacity(50);

**setLength():** Sets the length of the StringBuffer. If the length is reduced, characters are truncated. Example:

StringBuffer sb = new StringBuffer("Hello");

sb.setLength(3); // "Hel"

**charAt():** Returns the character at the specified index. Example:

StringBuffer sb = new StringBuffer("Hello");

char ch = sb.charAt(1); // 'e'

**substring():** Returns a substring from the specified start index to the end index (or end of the string). Example:

StringBuffer sb = new StringBuffer("Hello");

String sub = sb.substring(1, 3); // "el"

**public void setCharAt(int index, char ch):** Replaces the character at the specified index with the character ch.

**public int length():** Returns the current length of the StringBuffer (the number of characters it contains).

**trimToSize():** Reduces the capacity of the StringBuffer to its current length.

**→ The append() and insert()** methods in the StringBuffer class are overloaded. They can accept various data types as arguments, including int, boolean, long, and float.

**Advantages of StringBuffer:**

**--------------------------------------**

**Efficient for Modifications:** Avoids creating multiple objects in memory during string manipulations.

**Thread-Safe:** Suitable for multi-threaded applications due to synchronized methods.

**Flexible:** Provides a wide range of methods for string manipulation. Example:

public class StringBufferExample {

public static void main(String[] args) {

StringBuffer sb = new StringBuffer("Hello");

sb.append(" World"); // "Hello World"

sb.insert(5, " Java"); // "Hello Java World"

sb.replace(6, 10, "Programming"); // "Hello Programming World"

sb.reverse(); // "dlroW gnimmargorP olleH"

System.out.println(sb.toString());

}

}

**→ StringBuilder is functionally equivalent to StringBuffer,** with the key difference being that StringBuilder is not synchronized, while StringBuffer is synchronized. This means that StringBuilder is not thread-safe, but it offers better performance in single-threaded environments.

**StringBuffer vs StringBuilder:**

**Feature StringBuffer StringBuilder**

**------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Mutability** Mutable Mutable

**Thread Safety** Synchronized (thread-safe) Not synchronized (not thread-safe)

**Performance** Slower due to synchronization Faster due to lack of synchronization

overhead

**Use Case** Multi-threaded environments Single-threaded environments

**Introduced in** Java 1.0 Java 1.5

**→** Except for the above differences, everything else is the same in StringBuffer and StringBuilder, including methods and constructors.

**Feature String StringBuffer StringBuilder**

**-------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Definition**  Immutable sequence of characters. Mutable, thread-safe Mutable, non-thread-safe sequence

sequence of characters. of characters.

**Where to use?** When strings don't change When strings are modified When strings are modified

frequently. frequently in a multi-threaded frequently in a single-threaded

environment. environment for better performance.

**→** Many methods in the String, StringBuffer and StringBuilder classes return the same type (either String, StringBuffer, or StringBuilder). **This allows us to chain method calls**, where the result of one method call is used as the input for the next. In method chaining, the method calls are executed from left to right.

Case - 3:

------------

String s1 = "Hello Devi";

String s2 = "Hello" + " Devi";

System.out.println(s1 == s2); // Output: true

**( Explanation:** Because both "Hello" and " Devi" are string literals (constants), the compiler performs the concatenation at compile time. The compiler effectively replaces "Hello" + " Devi" with "Hello Devi". Therefore, both s1 and s2 are assigned the same string literal, "Hello Devi". When string literals are used, the JVM checks the String Constant Pool (SCP) first. Since "Hello Devi" is the same for both assignments, only one String object is created in the SCP. Both s1 and s2 then point to the same object in the SCP, resulting in s1 == s2 evaluating to true. This also applies to **final String variables** that are initialized with constant values.**)**

**→** String is a frequently used object in Java, which is why **Java provides specialized memory management for String objects in the form of the String Constant Pool (SCP).** This specialized memory management is not provided for StringBuffer objects.

**→** The SCP concept is available for String objects, which necessitates immutability. **Since multiple variables can point to the same String object in the SCP, changing the object's state would affect all variables referencing it.** StringBuffer objects are mutable because they are not stored in the SCP, and each StringBuffer variable typically points to its own unique object in the heap.

**( All wrapper class objects** in Java, like String objects, are immutable. The overloaded **+ and += operators** are also used for string concatenation. All immutable objects are inherently thread-safe because any attempt to modify their state results in the creation of a new object.**)**

**Wrapper Classes in Java:**

**----------------------------------**

Definition: Wrapper classes are used to convert primitive data types into objects and vice versa. They are part of the java.lang package and provide a way to use primitive data types as objects.

Purpose: Primitive data types (e.g., int, char, boolean) are not objects, so they cannot be used in contexts that require objects (e.g., collections like ArrayList). Wrapper classes allow primitive data types to be used as objects.

Types of Wrapper Classes: Java provides a wrapper class for each primitive data type:

Primitive Type Wrapper Class

------------------------------------------------------------------------

byte Byte

short Short

int Integer

long Long

float Float

double Double

char Character

boolean Boolean

Features of Wrapper Classes

--------------------------------------

**Immutable:** Wrapper classes are immutable, meaning their values cannot be changed once created. Any modification creates a new object.

Autoboxing and Unboxing:

------------------------------------

**Autoboxing:** Automatic conversion of primitive types to their corresponding wrapper classes. Example:

int num = 10;

Integer obj = num; // Autoboxing

**Unboxing:** Automatic conversion of wrapper class objects to their corresponding primitive types. Example:

Integer obj = 20;

int num = obj; // Unboxing

**Utility Methods:** Wrapper classes provide utility methods for converting between types, parsing strings, and more.

**Common Methods of Wrapper Classes:**

**-----------------------------------------------------**

**valueOf():** Converts a **primitive or string** into a wrapper class object. Example:

Integer num = Integer.valueOf(10); // int to Integer

Integer num2 = Integer.valueOf("20"); // String to Integer

**parseXxx():** Converts a string into a primitive type. Example:

int num = Integer.parseInt("30"); // String to int

**xxxValue():** Converts a wrapper class object into a primitive type. Example:

Integer obj = 40;

int num = obj.intValue(); // Integer to int

**toString():** Converts a wrapper class object or primitive type into a string. Example:

Integer obj = 50;

String str = obj.toString(); // Integer to String

**compareTo():** Compares two wrapper class objects numerically. Example:

Integer obj1 = 10;

Integer obj2 = 20;

int result = obj1.compareTo(obj2); // Returns -1 (obj1 < obj2)

**equals():** Checks if two wrapper class objects are equal. Example:

Integer obj1 = 10;

Integer obj2 = 10;

boolean isEqual = obj1.equals(obj2); // true

**Advantages of Wrapper Classes:**

**--------------------------------------------**

Allows primitive data types to be used as objects, enabling their use in collections like ArrayList, HashMap, etc. Provides useful methods for type conversion, parsing, and comparison. Wrapper classes can hold **null values**, whereas primitive types cannot. Enables compatibility with APIs that require objects.

**General Wrapper Class Constructors:**

**--------------------------------------------------**

Almost all numeric wrapper classes (e.g., Integer, Long, Double) **provide two constructors:** one that takes the corresponding primitive type as an argument, and another that takes a String argument. If the String argument does not represent a valid number for the given type, a NumberFormatException is thrown at runtime.

**Float Class:** The Float class is a bit of an exception, as it provides three constructors: one that takes a float, one that takes a double, and one that takes a String.

**Character Class:** The Character class has only one constructor, which takes a char argument.

**Boolean Class:** The Boolean class provides two constructors: one that takes a boolean primitive argument, and one that takes a String argument.

When using the boolean primitive constructor, the only valid arguments are true and false (case-sensitive). When using the string constructor, **if the string equals,** **ignoring case, to the string "true"**, then a true boolean object is created, otherwise a false boolean object is created. Example:

Boolean b1 = new Boolean("yes"); // false boolean object

Boolean b2 = new Boolean("no"); // false boolean object

Boolean b3 = new Boolean("True"); // true boolean object

System.out.println(b1.equals(b2)); // true

System.out.println(b1.equals(b3)); // false

**→** In wrapper classes, the **toString()** method is overridden to return the primitive value directly as a string and the **equals()** method is overridden for content comparison (i.e., comparing the primitive values).

Every wrapper class, except the Character class, contains a static valueOf() method to create a wrapper object from a given string. **Prototype: public static WrapperClass valueOf(String str);**

**→** Every integral type wrapper class (Byte, Short, Integer, Long) provides a valueOf(String s, int radix) method to create a wrapper object from a given string with a **specified radix.**

**Prototype: public static WrapperClass valueOf(String s, int radix);**

The allowed range for the radix parameter is from 2 to 36 (inclusive). This range is chosen because it allows for representations using the digits 0-9 (10 digits) and the letters A-Z (26 letters).

**→** Every numeric wrapper class (Byte, Short, Integer, Long, Float, Double) provides the following six instance methods to retrieve the primitive value of the wrapper object: byteValue(), shortValue(), intValue(), longValue(), floatValue(), and doubleValue(). Example:

Integer i = Integer.valueOf(130);

byte b = i.byteValue();

long l = i.longValue();

double d = i.doubleValue();

System.out.println(b); // -126

System.out.println(l); // 130

System.err.println(d); // 130.0

**→** The Character class contains the **charValue() method** to retrieve the char primitive value from a Character object. The Boolean class contains the **booleanValue()** method to retrieve the boolean primitive value from a Boolean object.

Example:

------------

Character ch = Character.valueOf('a');

char ch1 = ch.charValue();

Boolean b = Boolean.valueOf("Devi");

boolean b1 = b.booleanValue();

System.out.println(ch1+"......"+b1); // a......false

**→** Every numeric wrapper class (Byte, Short, Integer, Long, Float, Double), **except the Character class**, provides a static parseXXX() method to parse a string and return the corresponding primitive value. The prototype is: public static primitive parseXXX(String str);

Every integral type wrapper class (Byte, Short, Integer, Long) provides a static parseXXX(String s, int radix) method to parse a string representing an integer in a specified radix and return the corresponding primitive value. The prototype is: public static primitive parseXXX(String s, int radix); The allowed range for the radix parameter is from 2 to 36 (inclusive). Example:

int i = Integer.parseInt("1111", 2);

System.out.println(i); // 15 (Decimal of 1111)

**→** Every wrapper class contains the toString() method, which is an overriding version of the Object class's toString() method. This method converts the wrapper object to a String representation of its primitive value. Prototype: **public String toString();**

**→** Every wrapper class contains a static toString() method to convert a primitive value to a String representation. Prototype: public static String toString(primitive value); Example:

String str = Boolean.toString(false);

System.out.println(str); // false

**→ The Integer and Long classes** contain a static toString(primitive p, int radix) method to convert a primitive value to a String representation in a specified radix. The allowed range for the radix parameter is from 2 to 36 (inclusive).

Example: String s = Integer.toString(15, 2);

System.out.println(s); // 1111

**→** The Integer and Long classes provide the following static methods for converting primitive values to string representations in specific bases:

public static String **toBinaryString**(primitive p); (converts to base 2)

public static String **toOctalString**(primitive p); (converts to base 8)

public static String **toHexString**(primitive p); (converts to base 16)

Example: String str = Long.toHexString(10);

System.out.println(str); // a

**( Conclusion:** Boolean and Character are wrapper classes that do not inherit from the Number class. Byte, Short, Integer, Long, Float, and Double are all wrapper classes that extend the Number class, which in turn extends Object. Therefore they are indirect child classes of object.

String, StringBuffer, StringBuilder, and all wrapper classes are declared as final, meaning they cannot be subclassed. String and all wrapper classes are immutable. Once an instance is created, its state cannot be modified.**)**

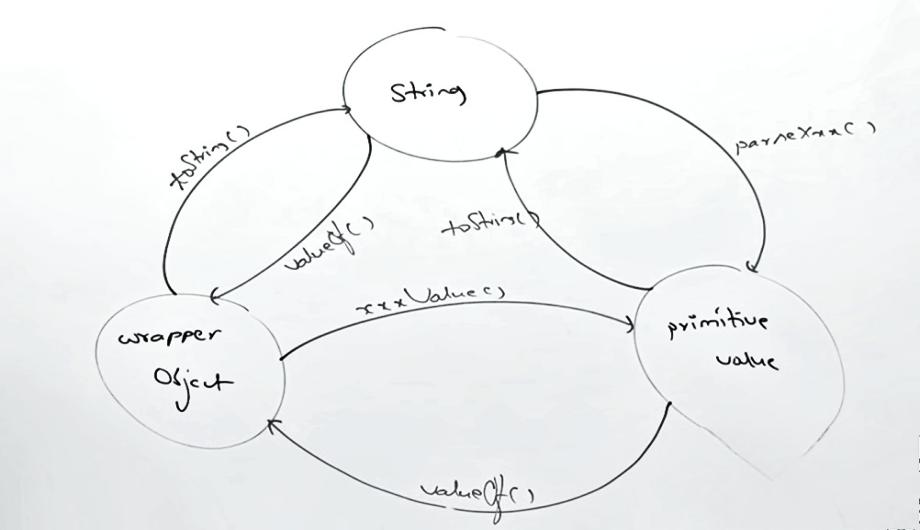
**→** Void is a final class that is a direct subclass of Object. It does not contain any instance methods. It contains only one **static final variable: Void.TYPE.** Void.TYPE represents the Class object for the void keyword. The Void class is commonly used in reflection to check whether a method's return type is void. For example:

if (getMethod("m1").getReturnType() == Void.TYPE) {

// Method m1 returns void

}

Essentially, Void serves **as the class representation of the void keyword** in Java.



**Autoboxing:** When you write Integer i = 10;, the compiler automatically inserts Integer i = Integer.valueOf(10); during compilation. This process, known as autoboxing, internally uses the valueOf() method of the wrapper class to convert the primitive value to a wrapper object. Autoboxing and autounboxing in Java were **introduced in Java 5 (JDK 5.0).**

**Autounboxing:** When you write int i1 = i;, the compiler automatically inserts int i1 = i.intValue(); during compilation. This process, known as autounboxing, internally uses the xxxValue() method (e.g., intValue(), doubleValue(), etc.) of the wrapper class to convert the wrapper object back to its corresponding primitive value.

**Case - 1:**

**------------**

public class Test {

static Integer i;

public static void main(String[] args) {

int i1 = i;

System.out.println(i1);

}

}

**Explanation:** ‘ i ’ is a static variable and is not explicitly initialized, it defaults to null. The compiler automatically inserts the intValue() method call, effectively making it ( int i1 = i.intValue(); ) This is autounboxing. Because ‘ i ’ is null, calling i.intValue() results in a **NullPointerException at runtime.**

**Case - 2:**

**------------**

Integer x = 10;

Integer y = x;

x++;

System.out.println(x); // 11

System.out.println(y); // 10

System.out.println(x == y); // false

**Explanation: x++:** This line increments the value of x. Because Integer objects are immutable, the increment operation

does not modify the existing Integer object. Instead, a new Integer object with the value 11 is created, and the variable x is reassigned to point to this new object. System.out.println(x == y); prints false because **x and y now point to different Integer objects** in memory.

**Case - 3:**

**------------**

Integer a = new Integer(10);

Integer b = 10;

Integer c = 10;

Integer d = 1000;

Integer e = 1000;

System.out.println(a == b); // false ( a points to an object created with new, while b points to a cached object )

System.out.println(b == c); // true ( true because both b and c point to the same cached object.)

System.out.println(d == e); // false ( false because d and e point to different objects created in the heap.)

**Explanation:**

**-----------------**

( Integer a = new Integer(10): This line explicitly creates a new Integer object in the heap using the new keyword. Therefore, this object is not retrieved from the cache.

Integer b = 10; and Integer c = 10 : These lines use autoboxing. The JVM checks the internal cache for an Integer object with the value 10. Since 10 is within the cached range (-128 to 127), both b and c point to the same cached Integer object.

Integer d = 1000; and Integer e = 1000: These lines also use autoboxing. However, 1000 is outside the cached range. Therefore, the JVM creates new Integer objects in the heap for both d and e. Thus, d and e point to different objects.)

**Wrapper Class Object Cache:**

----------------------------------------

To improve performance, Java maintains an internal cache of wrapper objects for certain ranges of values. This cache is initialized when the wrapper classes are loaded. When autoboxing occurs, the JVM first checks the cache. If the value is within the cached range, the JVM uses the cached object. If the value is outside the cached range, the JVM creates a new object. Cached Ranges:

Byte: Always cached (-128 to 127).

Short: Cached from -128 to 127.

Integer: Cached from -128 to 127 (can be configured in newer java versions).

Long: Cached from -128 to 127.

Character: Cached from 0 to 127.

Boolean: Always cached (both Boolean.TRUE and Boolean.FALSE).

Example: Boolean b1 = false;

Boolean b2 = false;

System.out.println(b1 == b2); // true

**→ No Cache for Float and Double:**  The internal cache for wrapper objects is not implemented for Float and Double. This is primarily due to the vast range of possible floating-point values and the complexity of caching them efficiently. Therefore, every autoboxing operation or valueOf() call for Float and Double will create a new object.

**valueOf() vs. new:** Since autoboxing internally uses the valueOf() method, the caching behavior is directly associated with valueOf(). It is generally recommended to use valueOf() instead of the new keyword when creating wrapper objects (especially for Byte, Short, Integer, Long, and Character) to take advantage of the caching mechanism and potentially improve performance. Using the new keyword will always create a new object. Example:

Integer i1 = 127;

Integer i2 = Integer.valueOf(127);

System.out.println(i1 == i2); // true

**Case - 4:**

**------------**

public class Test {

public static void m1(Integer num) {

System.out.println("Test.m1() - Integer");

}

public static void m1(long num) {

System.out.println("Test.m1() - Long");

}

public static void main(String[] args) {

int i = 10;

m1(i);

}

}

**Output:** Test.m1() - Long

**→** When the Java compiler resolves overloaded method calls, it follows a specific order of precedence to determine the best matching method. The order is as follows:

**Widening (Implicit Type Conversion):** Widening (also known as implicit type conversion or promotion) is preferred over other options.

**Autoboxing/Unboxing:** If no widening match is found, the compiler considers autoboxing or unboxing.

**Variable Arguments (Var-args):** Var-args has the lowest priority. If no widening or autoboxing/unboxing match is found, the compiler will consider var-args.

**(In summary:** Widening > Autoboxing/Unboxing > Var-args; If an exact match is found, that method is always chosen.**)**

**Case - 5:**

**------------**

public class Test {

public static void m1(Long num) {

System.out.println("Test.m1() - Integer");

}

public static void main(String[] args) {

int i = 10;

m1(i);

}

}

**( Explanation:** This code will result in a compile-time error (CE) because there's no valid method match for the argument type. Here's a concise explanation:

int cannot be widened directly to Long. Widening works for primitive to primitive types (e.g. int to long), not primitive to object wrapper types. The compiler can autobox int to Integer, but it cannot directly autobox int to Long. There is no implicit conversion path from int to Long that the compiler can use. Therefore, the compiler cannot find a suitable m1 method to invoke, leading to a compile-time error indicating that the method m1(int) is not applicable for the arguments.**)**

**Case - 6:**

**------------**

public class Test {

public static void m1(Object num) {

System.out.println("Test.m1() - Object");

}

public static void main(String[] args) {

int i = 10;

m1(i);

}

}

**Output:** Test.m1() - Object

**Explanation:**  **Autoboxing followed by widening is allowed in Java.** The compiler can first autobox a primitive value to its

corresponding wrapper object, and then widen the wrapper object reference to a superclass reference. **However, widening followed by autoboxing is not valid.** The compiler cannot directly widen a primitive value to a superclass type and then autobox it.

Example: int i = 10;

Number num = i; // Autoboxing followed by widening

System.out.println(num); // 10

**== Equality:**

**----------------**

If two objects are equal according to the == operator (i.e., they refer to the same object in memory), then they are always equal according to the equals() method (assuming the equals() method adheres to the general contract). If two objects are not equal according to the == operator, they may or may not be equal according to the equals() method. You cannot make any definitive prediction.

**equals() Equality:**

**------------------------**

If two objects are equal according to the equals() method, they may or may not be equal according to the == operator. You cannot make any definitive prediction. If two objects are not equal according to the equals() method, they are always not equal according to the == operator.

When you call the equals() method on a non-null object and pass null as an argument, the equals() method (if properly implemented) should return false.

**== Operator (Reference Equality):**

**----------------------------------------------**

To use the == operator for comparing object references, there must be a type relationship between the arguments. They must either be of the same type or have a parent-child (inheritance) relationship. If there is no such relationship (i.e., incompatible types), the compiler will generate a compile-time error indicating "incompatible types."

When you compare any object reference to null using the == operator, the result is false unless the object reference itself is null.

**equals() Method (Content Equality):**

**-------------------------------------------------**

The equals() method, as defined in the Object class, accepts an Object as its argument. Therefore, you can call equals() with objects of any type. If the objects are of incompatible types (i.e., they are not considered equal based on the equals() implementation), the equals() method will simply return false. It will not raise a compile-time error or a runtime exception (unless the equals() method itself throws an exception, which is generally bad practice).

**Feature == Operator equals() Method**

**-------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Definition & Applicability** Used for both primitives and objects Used only for objects (from Object class)

**Used for Comparison** Compares references for objects Compares content (if overridden) or

and values for primitives references (default implementation in

Object class)

**Can Override for Content** No, it always compares references for objects Yes, can be overridden for content

**Comparison?** comparison (e.g., in String, Wrapper classes)

**Comparing Unrelated Objects** No CE for primitives, but CE if two object No CE, just returns false if objects are

references have no relation unrelated

(not the same type or parent-child)

**( Objects.isNull() (Java 8 and later):** Java 8 introduced the Objects class, which includes the static method Objects.isNull(Object obj). This method provides a more readable alternative to obj == null. Example: if (Objects.isNull(myObject)) { ... } This method internally uses the == operator. It is mostly for readability. But the **== operator** is the most fundamental and efficient way to check for null in Java.**)**

**Hashing and Object Equality:**

----------------------------------------

Hashing-based data structures (like HashMap, HashSet) rely on the following principle: Two objects that are considered

equal (according to their equals() method) must have the same hash code. However, objects with the same hash code are not necessarily equal.

**equals() and hashCode() Contract:** The Object class's equals() and hashCode() methods adhere to this contract. Therefore, if you override the equals() method in your class, you must also override the hashCode() method to maintain this contract. If two objects are equal according to your overridden equals() method, their hashCode() methods must return the same value.

**Non-Equal Objects:** If two objects are not equal according to their equals() method, there is no restriction on their hash codes. They may or may not be the same.

**Equal Hash Codes:** If two objects have the same hash code, you cannot conclude anything about their equality. Their equals() method may return true or false.

**Unequal Hash Codes:**  If two objects have different hash codes, they are definitely not equal according to their equals() method.

**Importance of Overriding Together:** To maintain the contract between equals() and hashCode(), you must override hashCode() whenever you override equals(). Failing to do so will not result in a compile-time error or runtime exception. However, it will lead to incorrect behavior when using hashing-based data structures. This is considered poor programming practice.

**String Class:** The equals() method in the String class is overridden to compare the content of the strings. Consequently, the hashCode() method in the String class is also overridden to generate a hash code based on the content of the string. This ensures that two String objects with the same content will have the same hash code, as required by the equals()/hashCode() contract.

**StringBuffer Class:** The equals() method in the StringBuffer class is not overridden. It inherits the default equals() method from the Object class, which compares object references. Similarly, the hashCode() method in the StringBuffer class is also not overridden. It inherits the default hashCode() method from the Object class, which generates a hash code based on the object's reference. This means that even if two StringBuffer objects have the same content, they will not be considered equal by the equals() method unless they are the same object in memory. They will also likely have different hash codes.

**→** When you override the equals() method to compare objects based on specific parameters (e.g., fields or attributes), **it's crucial to override the hashCode() method using the same parameters.** This ensures that objects considered equal by equals() will also have the same hash code, which is essential for the proper functioning of hash-based data structures.

In Java's collection classes (like HashMap, HashSet), wrapper classes (like Integer, String), and the String class itself, the equals() method is overridden to perform content comparison. Therefore, it's generally good practice to override the equals() method in your own classes to compare objects based on their content, rather than their references.

**Cloning:** Cloning is the process of creating an exact copy of an object. The primary purpose of cloning is to create a backup or maintain a snapshot of an object's state.

**clone() Method:** The Object class provides a protected native Object clone() method that can be used for cloning. It throws a CloneNotSupportedException if the class does not implement the Cloneable interface. The clone method performs a shallow copy by default.

**Cloneable Interface:** To enable cloning for a class, the class must implement the Cloneable interface. The Cloneable interface is a marker interface, meaning it doesn't declare any methods. It simply signals to the JVM that the class supports cloning. Example:

public class Test implements Cloneable {

public static void main(String[] args) throws CloneNotSupportedException {

Test t1 = new Test();

Test t2 = (Test) t1.clone();

}

}

**Shallow vs. Deep Copy:**

**--------------------------------**

**Shallow copy:** Creates a new object, and copies the values of the original object's fields into the new object. If a field is a

reference to another object, only the reference is copied. Both the original and cloned objects will point to the same

referenced object.

**Deep copy:** Creates a new object, and recursively copies all referenced objects. This ensures that the cloned object has its own independent copies of all referenced objects.

**The Object.clone() method performs a shallow copy.** If you need a deep copy, you must override the clone() method and implement the deep copying logic yourself. Example:

public class Test {

public static void main(String[] args) throws CloneNotSupportedException {

Engine e1 = new Engine(1000);

Car c1 = new Car(e1, 2);

Car c2 = (Car) c1.clone();

System.out.println(c2.seats + "......" + c2.engine.hp); // 2......1000

c2.seats = 4;

c2.engine.hp = 1500;

System.out.println(c1.seats + "......" + c1.engine.hp); // 2......1000

System.out.println(c2.seats + "......" + c2.engine.hp); // 4......1500

}

}

class Engine {

int hp;

Engine(int hp) {

this.hp = hp;

}

}

class Car implements Cloneable {

Engine engine;

int seats;

Car(Engine engine, int seats) {

this.engine = engine;

this.seats = seats;

}

@Override

public Object clone() {

Engine e1 = new Engine(engine.hp);

Car c1 = new Car(e1, seats);

return c1;

}

}

**Output:** 2......1000

2......1000

4......1500

**(**If your object contains only primitive type variables, a shallow clone is generally sufficient. Since primitive types are copied by value, the cloned object will have its own independent copies of the primitive data. If your object contains reference variables (i.e., references to other objects), a deep clone is typically the best choice. This ensures that the cloned object has its own independent copies of all referenced objects, preventing unintended modifications to shared objects.**)**

**→** When you call intern() on a String object, it checks if a string with the same content already exists in the String Constant Pool (SCP). If a matching string exists in the SCP, intern() returns a reference to that SCP object. If a matching string does not exist, intern() adds the string to the SCP and returns a reference to the newly added SCP object. Example:

String s1 = new String("Java");

String s2 = s1.intern();

String s3 = "Java";

System.out.println(s1 == s2); // false

System.out.println(s2 == s3); // true

**Collections Framework**

**====================================================================================**

An array is an indexed collection of a fixed number of homogeneous data elements. The main advantage of arrays is that they allow us to store multiple values using a single variable, improving code readability.

**Limitations of Arrays:**

**-----------------------------**

**Fixed Size –** Once an array is created, its size cannot be changed dynamically. This means we must know the required size in advance, which may not always be possible.

**Homogeneous Elements Only –** Arrays can hold only elements of the same data type.

**No Built-in Methods for Operations –** Arrays are not based on a standard data structure (like a List or Set), so they lack built-in methods for common operations. Developers must implement functionalities like searching, sorting, and resizing manually, increasing programming complexity.

**Advantages of Collections Over Arrays:**

**-----------------------------------------------------**

To overcome the limitations of arrays, Java provides the Collections Framework, which offers dynamic, flexible, and efficient data structures. The key advantages of collections over arrays are:

**Dynamic Size –** Unlike arrays, collections are resizable. We can increase or decrease the size dynamically based on our requirements.

**Heterogeneous Elements –** Collections can store different types of objects (if generics are not used), whereas arrays strictly require homogeneous elements.

**Rich Built-in Methods –**  Collections provide ready-made methods for common operations like insertion, deletion,

searching, sorting, and iteration, reducing manual coding effort.

**Better Performance & Flexibility –** Collections like ArrayList, LinkedList, HashSet, etc. are optimized for different operations, allowing us to choose the best-suited data structure for a specific use case.

**Standardized Data Structures –**  Collections are implemented based on standard data structures (e.g., List, Set, Map), providing well-defined behaviors and efficient algorithms.

Thus, collections solve the fixed size, homogeneity, and lack of built-in method problems of arrays, making them more powerful and preferred in most real-world applications.

**Differences Between Arrays and Collections:**

**Feature Arrays Collections**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Static/Dynamic Size** Fixed in size (static) once created. Dynamic size, can grow or shrink as needed.

**Memory Efficiency** Can be memory-efficient if size is known. Uses more memory due to dynamic

resizing and object overhead.

**Performance Basis** Faster for direct access (index-based). Can be slower due to additional operations

like resizing, sorting, etc.

**Can Store Homogeneous/** Can store only homogeneous Can store Heterogeneous Elements

**heterogeneous** elements (same data type). elements (if generics are not used).

**Standardized Data Structures** Not based on standard data structures. Implemented using standard data

structures like List, Set, and Map.

**Used for Primitives/Objects** Can store both primitive types and objects. Can store only objects (wrapper

classes needed for primitives).

Collections are preferred when size is unknown or frequently changing, while arrays are useful when fixed-size storage with high performance is required.

**→** When you need to represent **a group of individual** **objects as a single unit, you should use a collection.** The Java **Collections Framework provides a set of interfaces and classes** that allow you to represent and manipulate groups of objects. These classes and interfaces are designed to offer various ways to store, retrieve, and process collections of data.

**9 Key Interfaces of the Collection Framework:**

**--------------------------------------------------------------**

**Collection:** The Collection interface defines the most common methods that are applicable to any collection object. It is an interface, and there are no concrete classes that directly implement the Collection interface. It is used to represent a group of individual objects as a single entity. **(The root interface of the collection hierarchy.)**

Common methods: add(), remove(), size(), isEmpty(), contains(), iterator().

**Sub-interfaces: List, Set, Queue.**

**Collections Utility Class:**

**---------------------------------**

Collections is a utility class (note the plural "s") present in the java.util package. It provides various static utility methods for working with collection objects, such as sorting, searching, and other operations. Collections is not an interface and is very different from the Collection interface.

**List:** Represents an ordered collection of elements (sequence). Allows duplicate elements. Elements can be accessed by their index. Common implementations: ArrayList, LinkedList, Vector.

Key methods: get(int index), set(int index, E element), add(int index, E element).

**Set:** Represents a collection of unique elements (no duplicates). Does not maintain any order (unless using LinkedHashSet or TreeSet). Common implementations: HashSet, LinkedHashSet, TreeSet.

Key methods: Inherits methods from Collection.

**SortedSet:** Extends the Set interface. Represents a set of elements sorted in natural order or by a specified comparator. Common implementation: TreeSet.

Key methods: first(), last(), headSet(), tailSet(), subSet().

**Vector and Stack (Java 1.2):** Prior to Java 1.2, Vector and Stack were part of the original Java Collections framework. In Java 1.2, with the introduction of the modern Collections Framework, these classes were retrofitted to implement the List interface. This allowed them to integrate more seamlessly with the new framework.

**NavigableSet:** NavigableSet is a subinterface of SortedSet. It extends SortedSet by providing methods for navigating the elements in the set, such as finding the closest match to a given value. TreeSet is the primary implementation class of the NavigableSet interface.

**Comparison between List and Set:**

**List Set**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Duplicates:** Allows duplicate elements.   Does not allow duplicate elements (elements are unique).

**Insertion Order:** Preserves the insertion order of elements.   Insertion order may or may not be preserved,

depending on the specific Set implementation (e.g.,

HashSet does not, LinkedHashSet does).

**Queue:** Represents a collection designed for holding elements **prior to processing.** Follows the FIFO (First-In-First-Out) principle, except for PriorityQueue. Common implementations: LinkedList, PriorityQueue, ArrayDeque.

Key methods: offer(), poll(), peek().

**Map:** Represents a collection of **key-value pairs.** Does not extend the Collection interface. **Keys are unique**, but values can be duplicated. Common implementations: HashMap, LinkedHashMap, TreeMap.

Key methods: put(), get(), remove(), keySet(), values(), entrySet().

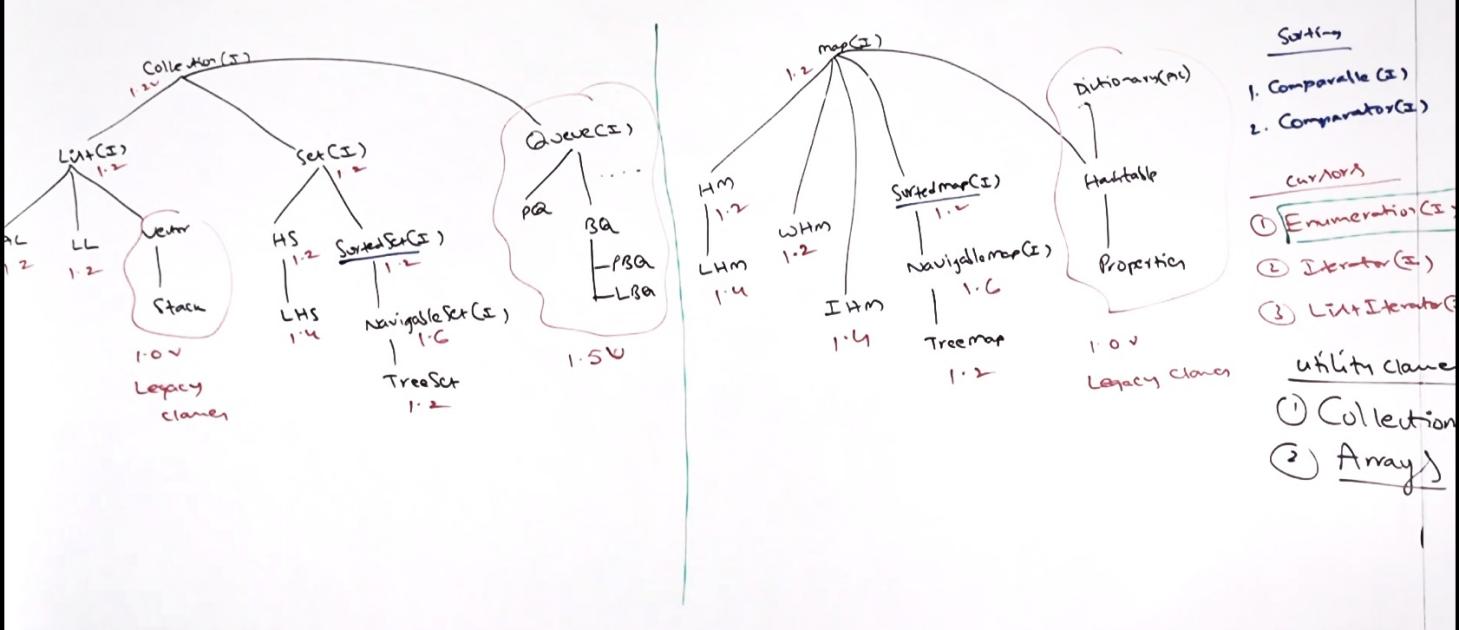
**SortedMap:** Extends the Map interface. Represents a map of key-value pairs **sorted by keys** in natural order or by a specified comparator. Common implementation: TreeMap.

Key methods: firstKey(), lastKey(), headMap(), tailMap(), subMap().

**→ NavigableMap is a subinterface of SortedMap.** It provides methods for navigating the entries in a map based on their keys. TreeMap is the primary implementation class of the NavigableMap interface.

**→ Legacy Components** of the Java Collections Framework: Enumeration (interface), Vector (class), Stack (class), Dictionary (abstract class), Hashtable (class), Properties (class)

These components were part of the original Java releases before the introduction of the more comprehensive Collections Framework in Java 1.2. While they are still available for backward compatibility, it's generally recommended to use the newer collection interfaces and classes for new development.



**Collection Interface in Java:**

**--------------------------------------**

**Definition:** The Collection interface is the root interface of the Java Collections Framework. It represents a group of objects, known as elements, and provides a unified way to store, retrieve, and manipulate these elements.

Purpose: It serves as the foundation for all collection types (e.g., List, Set, Queue). It defines the basic operations that all collections must support, such as adding, removing, and querying elements.

**Key Characteristics:** It is part of the java.util package. It does not have any **direct implementations;** it is implemented by its sub-interfaces (List, Set, Queue). It works with objects only (not primitive types).

**The Collection interface declares the following methods:**

**-----------------------------------------------------------------------------**

**Adding Elements: boolean add(E e):** Adds an element to the collection. Returns true if the collection changes as a result of the call.

**boolean addAll(Collection<? extends E> c):** Adds all elements from the specified collection to this collection.

**Removing Elements: boolean remove(Object o):** Removes a single instance of the specified element from the collection.

**boolean removeAll(Collection<?> c):** Removes all elements in the specified collection from this collection.

**boolean retainAll(Collection<?> c):** Retains only the elements in this collection that are also in the specified collection.

**void clear():** Removes all elements from the collection.

**Querying Elements:**

**---------------------------**

**boolean contains(Object o):** Checks if the collection contains the specified element.

**boolean containsAll(Collection<?> c):** Checks if the collection contains all elements of the specified collection.

**boolean isEmpty():** Checks if the collection is empty.

**int size():** Returns the number of elements in the collection.

**Iterating Over Elements: Iterator<E> iterator():**  Returns an iterator over the elements in the collection.

**Converting to Array: Object[] toArray():** Returns an array containing all elements in the collection.

**<T> T[] toArray(T[] a):** Returns an array containing all elements in the collection, with the runtime type of the specified array.

**Sub-Interfaces of Collection:** The Collection interface is extended by three main sub-interfaces: List, Set, Queue

**List Interface in Java:**

**-----------------------------**

**Definition:** The List interface is a sub-interface of the Collection interface. It represents an ordered collection of elements (also called a sequence). It allows duplicate elements and maintains the insertion order.

**Purpose:** It is used when the order of elements matters and duplicates are allowed. It provides methods to access elements by their index, making it easy to retrieve and manipulate elements at specific positions.

Key Characteristics:

---------------------------

**Ordered:** Elements are stored in a specific sequence.

**Indexed:** Elements can be accessed by their index (position).

**Allows Duplicates:** The same element can appear multiple times in the list.

**Allows null:** A List can store null values.

**Common Methods of the List Interface:**

**------------------------------------------------------**

The List interface extends the Collection interface and adds the following methods:

**Adding Elements: void add(int index, E element):** Inserts the specified element at the specified position.

**boolean addAll(int index, Collection<? extends E> c):** Inserts all elements from the specified collection into the list at the specified position.

**Removing Elements: E remove(int index):** Removes the element at the specified position.

**Updating Elements: E set(int index, E element):** Replaces the element at the specified position with the specified element.

**Accessing Elements: E get(int index):** Returns the element at the specified position.

**int indexOf(Object o):** Returns the index of the first occurrence of the specified element.

**int lastIndexOf(Object o):** Returns the index of the last occurrence of the specified element.

**Sublist Operations: List<E> subList(int fromIndex, int toIndex):** Returns a view of the portion of the list between the specified fromIndex (inclusive) and toIndex (exclusive).

**Iterating Over Elements: ListIterator<E> listIterator():** Returns a list iterator over the elements in the list.

**ListIterator<E> listIterator(int index):** Returns a list iterator starting at the specified position.

**→** The List interface is implemented by several classes in Java. The most commonly used ones are:

**ArrayList:** A resizable array implementation of the List interface. Provides fast random access and is efficient for retrieving elements by index. Not synchronized (not thread-safe).

**LinkedList:** A doubly-linked list implementation of the List interface. Provides efficient insertion and deletion of elements. Not synchronized (not thread-safe).

**Vector:** A synchronized (thread-safe) resizable array implementation of the List interface. Slower than ArrayList due to synchronization overhead.

**Stack:** A subclass of Vector that implements a last-in-first-out (LIFO) stack.

**ArrayList:**

**--------------**

**Underlying Data Structure:** Resizable array (growable array).

**Duplicates:** Duplicates are allowed.

**Insertion Order:** Insertion order is preserved.

**Heterogeneous Objects:** Heterogeneous objects are allowed (although generally not recommended for type safety).

**Null Insertion:** Null insertion is allowed.

**→** By default, most Java collections (including ArrayList, LinkedList, HashSet, HashMap, etc.) allow heterogeneous objects. However, **TreeSet and TreeMap** require objects to be comparable (either by implementing Comparable or providing a Comparator), which effectively restricts them to homogeneous or at least comparable types. It is very important to note that with the **introduction of generics, it is considered very bad practice to insert hetrogenous objects into collections.** Generics were added to java to provide type safety.

**Constructors of ArrayList in Java:**

**--------------------------------------------**

**ArrayList() –** Creates an empty list with an initial capacity of **10.**

**ArrayList(int capacity) –** Creates an empty list with the specified initial capacity.

**ArrayList(Collection<? extends E> c) –** Creates an ArrayList initialized with elements from the specified collection.

When an ArrayList reaches its current capacity, it grows by creating a new internal array with increased capacity. The growth formula is: **New capacity = (Current capacity × 3/2) + 1 (** However, this specific formula was used in older Java versions.**)**

In modern Java implementations: The default initial capacity is 10. **The growth factor is now 1.5 (or current capacity × 3/2) without the +1** (In **Java 8** and newer, the specific growth logic has been refined)

This resizing operation involves: Creating a new larger array. Copying all elements from the old array to the new one. Using the new array as the backing storage. This automatic resizing ensures ArrayList can grow dynamically, but may impact performance during resizing operations.

**Example:** ArrayList list = new ArrayList();

list.add("A");

list.add(10);

list.add("B");

list.add(null);

list.remove(2);

list.add(2, "D");

list.add(true);

System.out.println(list); **Output: [A, 10, D, null, true]**

**→** Collections are commonly used to hold and transfer objects between different parts of an application or across networks. To support this, most collection classes in the Java Collections Framework implement **the Serializable interface**, allowing them to be converted to a byte stream for storage or transmission. Many collection classes also **implement the Cloneable interface**, enabling them to create copies of themselves.

**→ ArrayList and Vector implement the RandomAccess interface.** This interface indicates that these classes support fast random access to their elements (i.e., accessing an element at any index takes constant time).

**RandomAccess Interface:** The RandomAccess interface is a marker interface located in the **java.util package.** It does not declare any methods. It serves as a signal to algorithms that the implementing class provides efficient random access.

**→ ArrayList is an excellent choice when retrieval is the primary operation.** Because it implements the RandomAccess interface, accessing elements by index is very efficient (constant time). **ArrayList is a poor choice when frequent insertions or deletions occur in the middle of the list.** Inserting or deleting an element in the middle requires shifting all subsequent elements, which can be time-consuming, especially for large lists.

**Difference Between ArrayList and Vector in Java:**

**Feature ArrayList Vector**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Synchronization** Not synchronized (not thread-safe). Synchronized (thread-safe).

**Performance** Faster because it does not have Slower due to synchronization overhead.

synchronization overhead.

**Thread Safety** Not thread-safe. Thread-safe.

**Growth Rate** Increases its size by 50% when it Doubles its size when capacity is exceeded.

exceeds capacity.

**Usage** Preferred for single-threaded applications. Preferred for multi-threaded applications.

**Legacy Status** Introduced in Java 1.2, part of the Java Introduced in Java 1.0, now considered

Collections Framework. legacy but still used.

**Enumeration Support** Uses Iterator and ListIterator for traversal. Supports both Enumeration (legacy) and Iterator.

**→** ArrayList is inherently non-synchronized (not thread-safe). You can obtain a synchronized (thread-safe) version of an ArrayList by using the synchronizedList() method of the **Collections utility class**. The prototype of the method is:

**public static List synchronizedList(List list); Example:**

ArrayList list = new ArrayList();

List l = Collections.synchronizedList(list);

**→** You can obtain a synchronized (thread-safe) version of a Set and Map by using the synchronizedSet() and synchronizedMap() methods of the Collections utility class.

**Prototype:** public static Set synchronizedSet(Set s);

public static Map synchronizedMap(Map m);

**LinkedList in Java:**

**-------------------------**

**Definition:** LinkedList is a class in Java that implements the List and Deque interfaces. It is part of the java.util package and represents a doubly-linked list data structure.

**Purpose:** It is used to store and manipulate a sequence of elements. It is particularly useful when frequent insertions and deletions are required, as it performs these operations efficiently.

Key Characteristics:

---------------------------

**Doubly-Linked List:** Each element (node) in the list contains a reference to the previous and next elements.

**Dynamic Size:** The size of the list can grow or shrink as needed.

**Non-Synchronized:** It is not thread-safe by default.

**Allows null:** A LinkedList can store null values.

Advantages of LinkedList:

-----------------------------------

**Efficient Insertions and Deletions:**  Adding or removing elements at the beginning, middle, or end of the list is efficient (O(1) for beginning and end, O(n) for middle).

**Flexible Operations:** Supports operations like adding/removing elements from both ends, making it suitable for implementing stacks, queues, and deques.

**No Wasted Memory:** Memory is allocated only when elements are added, unlike arrays which have a fixed size.

Disadvantages of LinkedList:

--------------------------------------

**Slower Access by Index:** Accessing elements by index is slower (O(n)) compared to ArrayList (O(1)).

**Higher Memory Overhead:** Each element requires additional memory for storing references to the previous and next

elements.

**→ LinkedList implements the Cloneable and Serializable interfaces** but does not implement the RandomAccess interface.

Constructors of the LinkedList class:

------------------------------------------------

**LinkedList():** This is the default constructor. It creates an empty LinkedList.

**LinkedList(Collection<? extends E> c):**  This constructor creates a LinkedList containing all of the elements in the specified collection, in the order they are returned by the collection's iterator.

**→** LinkedList is well-suited for implementing stacks and queues due to its efficient insertion and deletion operations at both ends of the list. Here are the methods it provides to support those implementations:

**Adding Elements:**

**------------------------**

**void addFirst(E e):** Adds an element to the beginning of the list.

**void addLast(E e):** Adds an element to the end of the list.

**Removing Elements:**

**-----------------------------**

**E removeFirst():** Removes and returns the first element of the list.

**E removeLast():** Removes and returns the last element of the list.

**Accessing Elements:**

**----------------------------**

**E getFirst():** Returns the first element of the list.

**E getLast():** Returns the last element of the list.

**Example:** LinkedList list = new LinkedList();

list.add("Java");

list.add(10);

list.add(false);

list.addFirst('A');

list.removeLast();

list.addLast(10.5);

list.set(0, 'D');

System.out.println(list); **Output:** [D, Java, 10, 10.5]

**Difference Between ArrayList and LinkedList:**

**Feature ArrayList LinkedList**

**--------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Insertion/Deletion** Slow, as shifting of elements is required. Fast, as only references are updated.

**Access Time (Search)** Fast (O(1) for direct access using index). Slow (O(n) as traversal is required).

**Memory Usage** Less, as it only stores actual elements. More, as it requires extra memory

for pointers.

**Best Used When** Frequent read/search operations. Frequent insert/delete operations.

**Performance** Faster for retrieving elements. Faster for modifying elements.

**Data Structure Used** Resizable Array Doubly Linked List

**Iteration Performance** Fast as it maintains an index-based structure. Slow as it requires node traversal.

**Adding Elements in Middle** Slow, as shifting is needed. Fast, as only node pointers are updated.

**Vector Class in Java:**

**---------------------------**

**Definition:**  The Vector class is a synchronized (thread-safe) implementation of the List interface. It is part of the java.util package and represents a dynamic array that can grow or shrink in size. Heterogeneous objects are allowed. Vector implements the Cloneable, Serializable, and RandomAccess interfaces. (Data Structure Used : **Resizable Array**)

**Purpose:** It is used to store and manipulate a sequence of elements. It is similar to ArrayList, but it is synchronized, making it suitable for use in multi-threaded environments.

Key Characteristics:

---------------------------

**Dynamic Array:** The size of the vector can grow or shrink as needed.

**Synchronized:** All methods in Vector are synchronized, making it thread-safe.

**Legacy Class:** It is part of the original Java Collections Framework (since Java 1.0).

**Allows null:** A Vector can store null values.

Disadvantages of Vector:

----------------------------------

**Performance Overhead:** Due to synchronization, it is slower than ArrayList in single-threaded environments.

**Less Flexible:** It lacks some of the newer features and optimizations available in ArrayList.

Vector class provides four constructors:

-----------------------------------------------------

**Vector() →** Creates a vector with default initial capacity of **10.**

**Vector(int initialCapacity) →** Creates a vector with **specified initial capacity.**

**Vector(int initialCapacity, int capacityIncrement) →** Creates a vector with a specified initial capacity and capacity

increment value.

**Vector(Collection<? extends E> c) →** Creates a vector containing all elements from the specified collection.

**→** When the capacity of a Vector is full and a new element is added, it expands based on the following formula:

**If capacityIncrement is specified:** New Capacity = Old Capacity + Capacity Increment

**If capacityIncrement is not specified (default behavior):** New Capacity = 2 × Old Capacity(Doubles the capacity)

**Vector-Specific Methods:**

**-----------------------------------**

**void addElement(E obj):** Adds an element to the end of the Vector.

**boolean removeElement(Object obj):** Removes the first occurrence of the specified object. Returns true if the element was

removed, otherwise false.

**void removeAllElements():** Removes all elements from the Vector.

**void removeElementAt(int index):** Removes the element at the specified index.

**E firstElement():** Returns the first element of the Vector. Throws NoSuchElementException if the Vector is empty.

**E lastElement():** Returns the last element of the Vector.Throws NoSuchElementException if the Vector is empty.

**Object elementAt(int index):** Returns the element at the specified index in the Vector.

Throws ArrayIndexOutOfBoundsException if the index is out of range.

**int capacity():** Returns the current capacity of the Vector.

**int size():** Returns the number of components (elements) in this vector.

**Enumeration elements():**  Returns an enumeration of the components of this vector.

Example: Vector v = new Vector<>();

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

v.addElement(i);

}

System.out.println(v.capacity()); // 10

v.addElement("Java");

System.out.println(v.capacity()); // 20

System.out.println(v); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, Java]

**→ Stack is a subclass of Vector** and is designed for **LIFO** (Last-In-First-Out) operations. The Stack class in Java is thread-safe because it extends Vector, and all its methods are synchronized.

Constructor Detail: **public Stack():** Creates an empty Stack.

Here are the Stack specific methods and their descriptions:

-------------------------------------------------------------------------------

**E push(E item):** Pushes an item onto the top of the stack. Returns the item pushed onto the stack.

**E pop():** Removes the item at the top of the stack and returns it. Throws EmptyStackException if the stack is empty.

**E peek():** Returns the item at the top of the stack without removing it. Throws EmptyStackException if the stack is empty.

**boolean empty():** Tests if the stack is empty. Returns true if the stack is empty, false otherwise.

**int search(Object o):** Returns the 1-based position of an object on the stack from the top. Returns -1 if the object is not on the stack.

**Example:** Stack s = new Stack<>();

s.push(10);

s.push(20);

int i = (Integer) s.pop();

System.out.println(i); // 20

s.push("Java");

System.out.println(s.peek()); // Java

System.out.println(s); // [10, Java]

System.out.println(s.search("A")); // -1

**→** To retrieve objects one by one from a collection, you use a cursor. Java provides three types of cursors: **Enumeration, Iterator, ListIterator**

**Enumeration:**

**-------------------**

Enumeration is a legacy interface introduced in Java 1.0. It is used to traverse elements of legacy collections like Vector and Hashtable. It provides a way to access elements sequentially but does not support modification of the collection during traversal. **The elements() method** of the Vector class is used to create an Enumeration object.

Key Methods:

-------------------

**boolean hasMoreElements():** Checks if there are more elements to iterate over.

**E nextElement():** Returns the next element in the iteration.

Example:

import java.util.Enumeration;

import java.util.Vector;

public class EnumerationExample {

public static void main(String[] args) {

Vector<String> vector = new Vector<>();

vector.add("Apple");

vector.add("Banana");

vector.add("Cherry");

Enumeration<String> enumeration = vector.elements();

while (enumeration.hasMoreElements()) {

System.out.print(enumeration.nextElement() + “ ”); // Apple Banana Cherry

}

}

}

**Limitations:** It is a read-only interface (cannot remove elements). **Only works with legacy classes like Vector and Hashtable**

**Iterator:** Iterator is a more modern interface introduced in Java 1.2. It is used to traverse collections like ArrayList, HashSet, LinkedList, etc. It supports both read and remove operations during traversal.

Key Methods:

-------------------

**boolean hasNext():** Checks if there are more elements to iterate over.

**E next():**  Returns the next element in the iteration.

**void remove():** Removes the last element returned by the iterator (optional operation). Example:

import java.util.ArrayList;

import java.util.Iterator;

public class IteratorExample {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String fruit = iterator.next();

System.out.print(fruit+” ”); // Apple Banana Cherry

if (fruit.equals("Banana")) {

iterator.remove(); // Remove "Banana" from the list

}

}

System.out.println("\nList after removal: " + list); // List after removal: [Apple, Cherry]

}

}

**Advantages:** Supports element removal during traversal. Works with all modern collection classes.

**Limitations:** Can only move forward **(unidirectional)**. Cannot add or replace elements.

**→** You can obtain an Iterator object by calling the **iterator() method of the Collection interface.** Prototype:

**public Iterator<E> iterator();**

**ListIterator**

**-----------------**

ListIterator is an enhanced version of Iterator introduced in Java 1.2. It is used specifically for List collections like ArrayList, LinkedList, etc. It supports **bidirectional traversal** (forward and backward) and **allows adding, replacing, and removing elements** during iteration.

**→** You can obtain a ListIterator object by calling the **listIterator() method of the List interface.** Prototype:

**public ListIterator<E> listIterator();**

**→ ListIterator is a subinterface of Iterator.** Therefore, all methods defined in the Iterator interface are implicitly available in the ListIterator interface.

Key Methods:

-------------------

**boolean hasNext():** Checks if there are more elements to iterate forward.

**E next():** Returns the next element in the iteration.

**boolean hasPrevious():** Checks if there are more elements to iterate backward.

**E previous():** Returns the previous element in the iteration.

**void add(E e):** Inserts an element into the list at the current position.

**void set(E e):** Replaces the last element returned by next() or previous().

**void remove():** Removes the last element returned by next() or previous().

**public int nextIndex():** Returns the index of the element that would be returned by a subsequent call to next(). Returns the list size if the list iterator is at the end of the list.

**public int previousIndex():** Returns the index of the element that would be returned by a subsequent call to previous(). Returns -1 if the list iterator is at the beginning of the list.

Example:

import java.util.ArrayList;

import java.util.ListIterator;

public class ListIteratorExample {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<>();

list.add("Apple");

list.add("Banana");

list.add("Cherry");

ListIterator<String> listIterator = list.listIterator();

System.out.print("Forward Traversal: ");

while (listIterator.hasNext()) {

String fruit = listIterator.next();

System.out.print(fruit + " ");

if (fruit.equals("Banana")) {

listIterator.set("Blueberry"); // Replace "Banana" with "Blueberry"

}

else if (fruit.equals("Apple")) {

listIterator.add("Orange");

}

}

System.out.println();

System.out.print("Backward Traversal: ");

while (listIterator.hasPrevious()) {

System.out.print(listIterator.previous()+" ");

}

System.out.println("\nList after modification: " + list);

}

}

**Output:** Forward Traversal: Apple Banana Cherry

Backward Traversal: Cherry Blueberry Orange Apple

List after modification: [Apple, Orange, Blueberry, Cherry]

**Advantages:** Supports bidirectional traversal. Allows adding, replacing, and removing elements during iteration.

**Limitations:** Only works with List collections.

**Comparison Table:**

**Feature Enumeration Iterator ListIterator**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------**

**Introduced in** Java 1.0 Java 1.2 Java 1.2

**Works with** Legacy collections (Vector, Hashtable) All modern collections List collections only

**Traversal Direction** Forward only Forward only Forward and backward

**Element Removal** No Yes Yes

**Element Addition** No No Yes

**Element Replacement** No No Yes

**Case - 1:**

**-------------**

Vector<String> v = new Vector<>();

Enumeration<String> enm = v.elements();

Iterator<String> i = v.iterator();

ListIterator<String> li = v.listIterator();

System.out.println(enm.getClass().getName()); // java.util.Vector$1

System.out.println(i.getClass().getName()); // java.util.Vector$Itr

System.out.println(li.getClass().getName()); // java.util.Vector$ListItr

**Explanation:** When we call elements(), iterator(), and listIterator() on a Vector, we don’t get the actual Iterator or Enumeration interfaces but instead receive objects of inner classes inside the Vector class that implement these interfaces. These inner classes are hidden implementations that allow Vector to provide different ways to traverse its elements.

**Set Interface:** The Set interface is a sub-interface of the Collection interface. It represents a collection of unique elements (no duplicates). It does not maintain any order unless using a specific implementation like LinkedHashSet or TreeSet.

**Key Characteristics:** No duplicate elements. Allows at most one null element. Does not provide index-based access to elements.

**Common Methods:** Inherits methods from Collection (e.g., add(), remove(), contains(), size())

**HashSet Class:** HashSet is a class that implements the Set interface. It uses a **hash table for storage** and provides constant-time performance for basic operations (add, remove, contains).

**Key Characteristics:** Does not maintain insertion order. **Allows one null element.** Not synchronized (not thread-safe).

**Constructors:**

**-------------------**

**HashSet():** Creates an empty HashSet with a **default initial capacity of 16 and a load factor of 0.75.**

**HashSet(int initialCapacity):** Creates an empty HashSet with the specified initial capacity.

**HashSet(int initialCapacity, float loadFactor):** Creates an empty HashSet with the specified initial capacity and load factor.

**HashSet(Collection<? extends E> c):** Creates a HashSet containing the elements of the specified collection. Example:

Set<String> set = new HashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Orange");

System.out.println(set); // [Apple, Banana, Orange] **(order may vary)**

**→** The Set interface implements Serializable and Cloneable, but not RandomAccess. HashSet is the most efficient choice for frequent search operations due to its underlying hash table implementation.

**→** HashSet is designed to store unique elements. It does not allow duplicates. Attempting to add a duplicate element to a HashSet does not result in a compile-time error (CE) or a runtime exception (RE). It simply returns false. add() Method returns false if the element was not added to the set (i.e., it was a duplicate). Example:

HashSet<Integer> set = new HashSet<>();

System.out.println(set.add(2)); // true

System.out.println(set.add(2)); // false

**→ Fill Ratio / Load Factor refers** to the threshold that determines when a rehashing (resizing) operation should occur in a HashMap, HashSet, or Hashtable.

**Definition:** Load Factor = (Current number of elements) / (Total capacity of the hash table)

**Default Load Factor:** In HashMap, the default load factor is 0.75 (75%). This means when the HashMap is 75% full, it resizes (doubles the capacity) to maintain efficiency.

**Example:** If a HashMap has a capacity of 16 and the load factor is 0.75, rehashing will happen when 16 × 0.75 = 12 elements are inserted.

**LinkedHashSet Class:** LinkedHashSet is a class that extends HashSet and implements the Set interface. **It maintains a linked list of the entries in the set,** preserving the insertion order.

Key Characteristics:

--------------------------

Maintains insertion order. Allows one null element. Not synchronized (not thread-safe).

**Constructors:**

**------------------**

**LinkedHashSet():** Creates an empty LinkedHashSet with a default initial capacity of 16 and a load factor of 0.75.

**LinkedHashSet(int initialCapacity):** Creates an empty LinkedHashSet with the specified initial capacity.

**LinkedHashSet(int initialCapacity, float loadFactor):** Creates an empty LinkedHashSet with the specified initial capacity and load factor.

**LinkedHashSet(Collection<? extends E> c):** Creates a LinkedHashSet containing the elements of the specified collection.

Example:

Set<String> set = new LinkedHashSet<>();

set.add("Apple");

set.add("Banana");

set.add("Orange");

System.out.println(set); // [Apple, Banana, Orange] (insertion order preserved)

**Feature HashSet LinkedHashSet**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Underlying Data Structure** HashMap (uses a Hash Table) LinkedHashMap (Hash Table + Doubly Linked List)

**Insertion Order Preserved?** No (Unordered) Yes (Maintains insertion order)

**Introduced In** Java 1.2 Java 1.4

**LinkedHashSet for Cache-Based Applications:**

**-------------------------------------------------------------**

LinkedHashSet is an excellent choice for implementing cache-based applications where: Duplicate elements are not allowed (maintaining uniqueness). The insertion order of elements needs to be preserved (maintaining recency or access order).

**SortedSet Interface:**

**---------------------------**

SortedSet is a sub-interface of the Set interface. It represents a set of elements sorted in natural order or by a specified comparator.

**Key Characteristics:** Elements are sorted in ascending order. Provides methods to access the first and last elements, and to retrieve subsets.

**Common Methods:**

**---------------------------**

**E first():** Returns the first (lowest) element.

**E last():** Returns the last (highest) element.

**SortedSet<E> headSet(E toElement):** Returns a view of the portion of the set whose elements are less than toElement.

**SortedSet<E> tailSet(E fromElement):**  Returns a view of the portion of the set whose elements are **greater than or equal to** fromElement.

**SortedSet<E> subSet(E fromElement, E toElement):** Returns a view of the portion of the set whose elements range from fromElement (inclusive) to toElement (exclusive).

**Comparator comparator():** Returns the Comparator used to order the elements in this sorted set, or **null if this set uses the natural ordering** of its elements.

The default natural sorting order is: Numbers: Ascending order (from smallest to largest).

Strings: Alphabetical order (lexicographical order).

**NavigableSet Interface:**

**--------------------------------**

NavigableSet is a sub-interface of SortedSet. It provides additional methods for navigation (e.g., finding the closest match for a given element).

**Key Characteristics:** Extends SortedSet with methods for retrieving elements based on the closest match. Allows reverse traversal of the set.

Common Methods:

--------------------------

**E lower(E e):** Returns the greatest element less than e.

**E floor(E e):** Returns the greatest element less than or equal to e.

**E ceiling(E e):** Returns the smallest element greater than or equal to e.

**E higher(E e):** Returns the smallest element greater than e.

**Iterator<E> descendingIterator():** Returns an iterator over the elements in descending order.

**TreeSet Class:**

**-------------------**

TreeSet is a class that implements the NavigableSet interface. **It uses a Red-Black Tree** for storage and maintains elements in sorted order.

**Key Characteristics:**

Maintains elements in sorted order (natural order or by a comparator). Does not allow null elements (if natural ordering is used). Not synchronized (not thread-safe). TreeSet implements the Serializable and Cloneable interfaces but not the RandomAccess interface.

Constructors:

------------------

**TreeSet():** Creates an empty TreeSet sorted according to the natural ordering of its elements.

**TreeSet(Comparator<? super E> comparator):** Creates an empty TreeSet sorted according to the specified comparator.

**TreeSet(Collection<? extends E> c):** Creates a TreeSet containing the elements of the specified collection, sorted according to natural ordering.

**TreeSet(SortedSet<E> s):** Creates a TreeSet containing the elements of the specified sorted set. Example:

Set<String> set = new TreeSet<>();

set.add("Apple");

set.add("Banana");

set.add("Orange");

System.out.println(set); // [Apple, Banana, Orange] (sorted order)

**→** If you attempt to insert heterogeneous objects into a TreeSet, the TreeSet will try to compare them. If the objects are not of comparable types (i.e., they cannot be compared using compareTo() or the Comparator), a **ClassCastException will be thrown at runtime.** This exception occurs because the TreeSet cannot determine the relative order of the objects.

**→** If a TreeSet already contains elements, attempting to insert null will result in a **NullPointerException (NPE).** This is because TreeSet relies on comparisons (either natural ordering or a Comparator), and null cannot be compared. In some older Java versions, it was **possible to insert null as the first element** into an empty TreeSet. However, even in those cases, subsequent attempts to insert any other element would then result in an NPE. **In modern java versions, even inserting null as the first element will result in a NPE.**

**Natural Ordering and Comparability:**

**--------------------------------------------------**

When relying on the default natural sorting order in Java (e.g., in TreeSet or TreeMap), the objects being sorted must be:

**Homogeneous:**  They must be of the same type or of types that are mutually comparable.

**Comparable:** They must implement the Comparable interface.

If these conditions are not met, **a ClassCastException (CCE)** will be thrown at runtime. An object is considered comparable if its class **implements the Comparable interface.** This interface defines the **compareTo() method**, which is used to determine the relative order of two objects. The String class, all wrapper classes (e.g., Integer, Double), and many other standard Java classes implement the Comparable interface.

**→** The Comparable interface is located in the java.lang package. It defines a single method: **public int compareTo(Object obj);**

compareTo() Mechanism:

-----------------------------------

The compareTo() method is used to compare the current object (this) with the specified object (obj). It returns an int value that indicates the relative order of the two objects:

**A negative integer** (e.g., -1) if this object is less than obj.

**Zero (0)** if this object is equal to obj.

**A positive integer** (e.g., 1) if this object is greater than obj.

The implementation of compareTo() defines the natural ordering of the objects. The mechanism is used by sorting algorithms, and data structures like treeset and treemap to order the elements. Example:

System.out.println("A".compareTo("Z")); // -25

System.out.println("Z".compareTo("A")); // 25

System.out.println("A".compareTo("A")); // 0

**→** When you add objects to a TreeSet that relies on the default natural sorting order, **the JVM will internally call the**

**compareTo()** method of the objects. The call is made in the form of obj1.compareTo(obj2), where: obj1 is the object being inserted.obj2 is an object already present in the TreeSet. This comparison determines the correct position of obj1 within the sorted TreeSet.

**→** If the default natural sorting order is not available (e.g., the objects don't implement Comparable) or **if you need a different sorting order, you can provide a custom Comparator to the TreeSet constructor.** This allows you to define your own sorting logic.

**Comparator Interface:**

**------------------------------**

The Comparator interface is located in the java.util package. It defines two primary methods:

**public int compare(Object obj1, Object obj2):** This method compares two objects (obj1 and obj2) and returns an int value indicating their relative order (negative, zero, or positive). The compare() method defines the customized sorting logic.

**public boolean equals(Object obj):** While the Comparator interface declares the equals(Object obj) method, it's important to understand that it's typically not explicitly implemented by Comparator implementations. The equals() method that is being called is the object equals method that is inherited from the object class. It is generally not necessary to override the equals method.

**compare() Method Mechanism:**

**-------------------------------------------**

**It returns:** A negative integer if obj1 is less than obj2.

Zero if obj1 is equal to obj2.

A positive integer if obj1 is greater than obj2

**Example - 1:**

**-----------------**

import java.util.Comparator;

import java.util.TreeSet;

public class Test {

public static void main(String[] args) {

TreeSet<Integer> ts = new TreeSet<>(new MyComparator());

ts.add(10);

ts.add(0);

ts.add(20);

ts.add(50);

ts.add(5);

System.out.println(ts); // [50, 20, 10, 5, 0]

}

}

class MyComparator implements Comparator<Integer> {

@Override

public int compare(Integer i1, Integer i2) {

if (i1 < i2) {

return +1;

}

else if (i1 > i2) {

return -1;

}

else {

return 0;

}

}

}

**→** In the line **TreeSet<Integer> ts = new TreeSet<>(new MyComparator());**, you are explicitly providing a Comparator object (new MyComparator()) to the TreeSet constructor. In this case, the TreeSet will use the **compare()** method of your MyComparator to determine the order of elements.

If you create a TreeSet without providing a Comparator, the TreeSet will rely on the natural ordering of the elements. This means that the elements must implement the Comparable interface, and the TreeSet will internally call the compareTo() method of the elements to determine their order.

**→** Let's analyze each of those return statements within the compare method of MyComparator and how they affect the element insertion order in a TreeSet<Integer>.  **Analysis:**

**return i1.compareTo(i2):** This is the standard ascending order. The TreeSet will maintain elements in their natural ascending order (e.g., 1, 2, 3, ...).

**return i2.compareTo(i1):** This reverses the natural order. The TreeSet will maintain elements in descending order (e.g., 3, 2, 1, ...).

**return -i1.compareTo(i2):** This also reverses the natural order, but using negation. The TreeSet will maintain elements in descending order (e.g., 3, 2, 1, ...).

**return -i2.compareTo(i1):** This also results in the natural ascending order, but using negation. The TreeSet will maintain elements in ascending order (e.g., 1, 2, 3, ...).

**return +1:** This forces all elements to be considered "greater than" the previous one. The TreeSet will maintain the elements in the order they are inserted. Crucially, all elements will be stored, and **no element will be treated as a duplicate.** The TreeSet will effectively behave like a LinkedList in terms of insertion order.

**return -1:** This forces all elements to be considered "less than" the previous one. The TreeSet will maintain the elements in the reverse of the order they are inserted. Again, all elements will be stored, and no element will be treated as a duplicate.

**return 0:** This forces all elements to be considered **"equal"**. The TreeSet will only store the first element that is added. All subsequent element additions will be treated as duplicates and ignored.

**Example - 2:**

**-----------------**

import java.util.Comparator;

import java.util.TreeSet;

public class Test {

public static void main(String[] args) {

TreeSet<String> ts = new TreeSet<>(new MyComparator());

ts.add("Apple");

ts.add("Ball");

ts.add("Cat");

ts.add("Devil");

System.out.println(ts); // [Devil, Cat, Ball, Apple]

}

}

class MyComparator implements Comparator<String> {

@Override

public int compare(String s1, String s2) {

return s2.compareTo(s1);

}

}

**→** While you can technically add heterogeneous objects if your Comparator is designed to handle them, it's generally considered bad practice. It can lead to complex and error-prone code. It is possible to write a comparator that handles heterogenous objects, but it is not recommended.

**Example - 3:**

**-----------------**

import java.util.Comparator;

import java.util.TreeSet;

public class Test {

public static void main(String[] args) {

TreeSet<Object> ts = new TreeSet<>(new MyComparator());

ts.add("Apple");

ts.add("Ball");

ts.add("Cat");

ts.add(new StringBuffer("Devil"));

ts.add(new StringBuffer("Elephant"));

System.out.println(ts); // [Cat, Ball, Apple, Devil, Elephant]

}

}

class MyComparator implements Comparator<Object> {

@Override

public int compare(Object o1, Object o2) {

String s1 = o1.toString();

String s2 = o2.toString();

if (s1.length() < s2.length()) {

return -1;

}

else if (s1.length() > s2.length()) {

return +1;

}

else {

return s1.compareTo(s2);

}

}

}

The above code is an example demonstrating the comparison of two heterogeneous objects (String, StringBuffer) and their sorting according to string length. If the lengths are identical, the objects are then sorted alphabetically.

**Predefined Comparable Classes:** These classes (e.g., String, Integer) have a built-in natural sorting order defined by their implementation of the Comparable interface. If you need a different sorting order, you can provide a custom Comparator.

**Predefined Non-Comparable Classes:** Classes like StringBuffer (before Java 21) don't implement Comparable, so they don't have a natural sorting order. You must use a Comparator to sort instances of these classes.

**User-Defined Classes (e.g., Employee):** As the class author, you can define the natural sorting order by implementing the Comparable interface. This allows users of your class to sort objects using the default natural ordering. However, users can still provide a custom Comparator if they need a different sorting order than the one you defined. Example:

import java.util.Comparator;

import java.util.TreeSet;

public class Test {

public static void main(String[] args) {

TreeSet<Employee> ts = new TreeSet<>(new MyComparator());

ts.add(new Employee(1002, "Prasad"));

ts.add(new Employee(1003, "Rahul"));

ts.add(new Employee(1001, "Devi"));

ts.add(new Employee(1000, "Sanket"));

System.out.println(ts); // [1001.....Devi, 1002.....Prasad, 1003.....Rahul, 1000.....Sanket]

}

}

class Employee implements Comparable<Object> {

int eID;

String eName;

Employee(int eID, String eName) {

this.eID = eID;

this.eName = eName;

}

@Override

public int compareTo(Object e) {

Integer id1 = this.eID;

Employee e1 = (Employee) e;

Integer id2 = e1.eID;

return id1.compareTo(id2);

}

@Override

public String toString() {

return this.eID + "....." + this.eName;

}

}

class MyComparator implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

return e1.eName.compareTo(e2.eName);

}

}

**Feature Comparable Comparator**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Definition** Used to define natural ordering of objects. Used to define custom ordering of objects.

**Package** java.lang java.util

**Method** Implements compareTo(Object o) method. Implements compare(Object o1, Object o2)

method.

**Modifying Sorting Logic** Cannot modify sorting logic without Allows multiple sorting logic by creating

changing the class. different comparator classes.

**Usage** Implemented by the class whose Used when we want to sort objects of a

objects need to be sorted. class but don’t modify the class itself.

**Feature HashSet LinkedHashSet TreeSet**

**--------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Underlying Data Structure** HashTable HashTable + LinkedList Red-Black Tree

(Backed by HashMap) (Backed by LinkedHashMap) (Self-balancing BST)

**Duplicates Allowed?** No No No

**Insertion Order Preserved?** No Yes No (Sorted Order)

**Sorting Order** No Sorting No Sorting Natural Ordering / Custom Comparator

**Heterogeneous Objects** Yes Yes No (Causes ClassCastException)

**Allowed?**

**Null Acceptance?** Allows a single null Allows a single null Does not allow null (Throws NullPointerException)

**Map Interface in Java:**

**-------------------------------**

The Map interface is part of the java.util package and represents a collection of key-value pairs where both keys and values are objects. Each key in a Map is unique, and it maps to exactly one value.

**Purpose:** It is used to store and manipulate data in the form of key-value pairs. It is widely used for lookups, where you can retrieve a value based on its key.

**Key Characteristics:**

**Unique Keys:** No duplicate keys are allowed.

**One-to-One Mapping:** Each key maps to exactly one value.

**Allows null:** Some implementations allow null keys and/or null values.

**Not a Subtype of Collection:** The Map interface is separate from the Collection interface.

**→** Each key-value pair is referred to as an **"entry"**. Therefore, a Map can be viewed as a collection of "entry" objects. The Map interface itself contains the **Entry interface**, which represents a key-value pair.

The Map interface provides the following key methods:

**Adding Elements:**

**-------------------------**

**V put(K key, V value):** Adds a key-value pair to the map. If the key already exists, the value is replaced.

**void putAll(Map<? extends K, ? extends V> m):** Adds all key-value pairs from the specified map to this map.

**Removing Elements: V remove(Object key):** Removes the key-value pair for the specified key.

**void clear():** Removes all key-value pairs from the map.

**Accessing Elements: V get(Object key):** Returns the value associated with the specified key.

**boolean containsKey(Object key):** Checks if the map contains the specified key.

**boolean containsValue(Object value):** Checks if the map contains the specified value.

**Size and Status: int size():** Returns the number of key-value pairs in the map.

**boolean isEmpty():** Checks if the map is empty.

**Views: Set<K> keySet():** Returns a Set view of all keys in the map.

**Collection<V> values():** Returns a Collection view of all values in the map.

**Set<Map.Entry<K, V>> entrySet():** Returns a Set view of all key-value pairs (entries) in the map.

**→** If the specified key is already present in the Map, the value associated with that key is replaced with the new value. In this case, the **put() method returns the old value** that was previously associated with the key. If the specified key is not present in the Map, the key-value pair is inserted into the Map. In this case, the put() method returns null.

**Entry Interface:**

**---------------------**

The Map.Entry interface represents a single key-value pair in a Map. It is a nested interface within the Map interface.

Common methods:

--------------------------

**K getKey():** Returns the key.

**V getValue():** Returns the value.

**V setValue(V value):** Replaces the value associated with the key.

The Map interface is implemented by several classes in Java. The most commonly used ones are:

**HashMap:** A hash table-based implementation of the Map interface.Allows null keys and null values. Not synchronized (not thread-safe). Example:

Map<String, Integer> map = new HashMap<>();

map.put("Apple", 1);

map.put("Banana", 2);

System.out.println(map.get("Apple")); // 1

**LinkedHashMap:** Extends HashMap and maintains insertion order. Allows null keys and null values. Not synchronized (not

thread-safe). Example:

Map<String, Integer> map = new LinkedHashMap<>();

map.put("Apple", 1);

map.put("Banana", 2);

System.out.println(map); // {Apple=1, Banana=2} (insertion order preserved)

**TreeMap:** A Red-Black Tree-based implementation of the SortedMap interface. Maintains elements in sorted order (natural order or by a comparator). Does not allow null keys (if natural ordering is used). Not synchronized (not thread-safe). **Ex:**

Map<String, Integer> map = new TreeMap<>();

map.put("Apple", 1);

map.put("Banana", 2);

System.out.println(map); // {Apple=1, Banana=2} (sorted order)

**Hashtable:** A legacy synchronized implementation of the Map interface. **Does not allow null keys or null values.** Thread-safe but slower than HashMap. Example:

Map<String, Integer> map = new Hashtable<>();

map.put("Apple", 1);

map.put("Banana", 2);

System.out.println(map.get("Banana")); // 2

**(HashMap is the most commonly used implementation due to its performance and flexibility.)**

**HashMap in Java:**

**------------------------**

HashMap is a class in Java that implements the Map interface. It stores data in key-value pairs and uses a hash table for storage. It is part of the java.util package.

**Purpose:** It is used to store and retrieve data efficiently based on keys. It provides constant-time performance (O(1)) for basic operations like get and put, assuming the hash function distributes elements properly.

Key Characteristics:

---------------------------

**Unique Keys:** No duplicate keys are allowed.

**Allows null:** It allows one null key and multiple null values.

**Unordered:** It does not maintain any order of elements.

**Not Synchronized:** It is not thread-safe by default.

How HashMap Works?

-------------------------------

**Hashing:** HashMap uses a hash function to compute an index (hash code) for each key. This index determines where the key-value pair is stored in the internal array (called buckets).

**Buckets:** The internal array is divided into buckets. Each bucket can store multiple key-value pairs (in case of hash collisions).

**Handling Collisions:** If two keys have the same hash code (collision), they are stored in the same bucket as a linked list or a balanced tree (in Java 8+).

**Load Factor:** The load factor is a measure of how full the HashMap is allowed to get before its capacity is automatically increased. The default load factor is 0.75.

Constructors of HashMap:

------------------------------------

**HashMap():** Creates an empty HashMap with a default initial capacity of **16 and a load factor of 0.75.** Example: HashMap<String, Integer> map = new HashMap<>();

**HashMap(int initialCapacity):** Creates an empty HashMap with the specified initial capacity and a load factor of 0.75.

Example: HashMap<String, Integer> map = new HashMap<>(32);

**HashMap(int initialCapacity, float loadFactor):** Creates an empty HashMap with the specified initial capacity and load factor.

Example: HashMap<String, Integer> map = new HashMap<>(32, 0.6f);

**HashMap(Map<? extends K, ? extends V> m):** Creates a HashMap containing the key-value pairs of the specified map.

Example: Map<String, Integer> oldMap = new HashMap<>();

oldMap.put("Apple", 1);

HashMap<String, Integer> map = new HashMap<>(oldMap);

**→** It is not synchronized (not thread-safe). For thread-safe operations, use **Collections.synchronizedMap() or ConcurrentHashMap.**

**→** HashMap implements the **Serializable and Cloneable** interfaces but does not implement the RandomAccess interface. HashMap is highly optimized for search operations due to its underlying hash table implementation, making it the preferred choice when frequent searches are required. Example:

import java.util.\*;

public class Test {

public static void main(String[] args) {

HashMap<Integer, String> map = new HashMap<>();

map.put(1, "Java");

map.put(2, "Python");

map.put(3, "C#");

map.put(4, "JavaScript");

System.out.println(map);

System.out.println(map.put(4, "TypeScript"));

Set<Integer> set = map.keySet();

System.out.println(set);

Collection<String> list = map.values();

System.out.println(list);

Set<Map.Entry<Integer, String>> set1 = map.entrySet();

System.out.println(set1);

Iterator<Map.Entry<Integer, String>> i1 = set1.iterator();

while (i1.hasNext()) {

Map.Entry<Integer, String> entry = (Map.Entry<Integer, String>) i1.next();

System.out.println(entry.getKey() + "........" + entry.getValue());

if (entry.getKey().equals(3)) {

entry.setValue("C++");

}

}

System.out.println(map);

}

}

**Output:** {1=Java, 2=Python, 3=C#, 4=JavaScript}

JavaScript

[1, 2, 3, 4]

[Java, Python, C#, TypeScript]

[1=Java, 2=Python, 3=C#, 4=TypeScript]

1........Java

2........Python

3........C#

4........TypeScript

{1=Java, 2=Python, 3=C++, 4=TypeScript}

**Feature HashMap Hashtable**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Synchronization Not synchronized Synchronized

Thread-safety Not thread-safe Thread-safe

Performance Faster (No synchronization overhead) Slower (Due to synchronization)

Null Insertion Allows one null key and multiple null values Does not allow null key or null Values

Introduced In Java 1.2 Java 1.0

**LinkedHashMap in Java:**

**---------------------------------**

**Definition:** LinkedHashMap is a class in Java that extends HashMap and implements the Map interface. It maintains a linked list of the entries in the map, preserving the insertion order or access order of the elements.

**Purpose:** It is used to store and retrieve data in key-value pairs while maintaining the order of insertion or access. It combines the benefits of HashMap (fast lookups) with predictable iteration order.

**Underlying Structure:** Uses a hash table internally, like HashMap, but also maintains a doubly-linked list that runs through all of its entries. This linked list defines the iteration ordering, which is normally the order in which keys were inserted into the map (insertion-order).

**→** It allows one null key and multiple null values. It is not thread-safe by default.

**Comparison between HashMap and LinkedHashMap:**

**Feature HashMap LinkedHashMap**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Underlying Structure** Hash Table Hash Table + Doubly-Linked List

**Insertion Order** Not Preserved **Preserved**

**Java Version** 1.2 1.4

**Performance (Basic)** Faster Slightly Slower

**Use Cases** General, Order Not Important **Caches**, Order Important

**Iteration Order**  Undefined Insertion or Access Order

**Example:**  LinkedHashMap<Integer, String> map = new LinkedHashMap<>();

map.put(1, "Java");

map.put(2, "C++");

map.put(3, "React");

System.out.println(map); // **{1=Java, 2=C++, 3=React}**

**→** LinkedHashSet and LinkedHashMap are indeed frequently used for developing cache-based applications.

**The key difference between HashMap and IdentityHashMap:**

**-----------------------------------------------------------------------------------**

**HashMap: Uses the equals() method** to determine key equality. This means that two keys are considered equal if their content is the same, according to the equals() method's implementation.

**IdentityHashMap: Uses the == operator** to determine key equality. This means that two keys are considered equal only if they refer to the same object instance in memory (i.e., they have the same memory address).

**Implications:** IdentityHashMap can store multiple key-value pairs where the keys have the same content but are different object instances. HashMap will treat those keys as duplicates and replace the old value with the new value.

**( All other methods and constructors are the same.)**

Example: IdentityHashMap<Integer, String> map = new IdentityHashMap<>();

Integer i1 = new Integer(10);

Integer i2 = new Integer(10);

map.put(i1, "Java");

map.put(i2, "C++");

System.out.println(map); **// {10=Java, 10=C++}**

**Example:**

import java.util.WeakHashMap;

public class Test {

public static void main(String[] args) throws InterruptedException {

WeakHashMap<Temp, String> map = new WeakHashMap<>();

Temp t1 = new Temp();

map.put(t1, "Java");

System.out.println(map);

t1 = null;

System.gc();

Thread.sleep(3000);

System.out.println(map);

}

}

class Temp {

@Override

public String toString() {

return "Temp";

}

@Override

public void finalize() {

System.out.println("Finalize method called");

}

}

**Output:** {Temp=Java}

Finalize method called

{}

**→** In a HashMap, if a key object is still referenced by the HashMap, it will not be garbage collected, even if there are no other references to it in the application. Essentially, the **HashMap holds strong references to its keys**, preventing them from being garbage collected. **This means that the HashMap dominates the Garbage collector.**

**In a WeakHashMap, the keys are held using weak references.** If a key object is no longer strongly referenced by any part of the application (outside of the WeakHashMap), it becomes eligible for garbage collection. When the garbage collector reclaims the key object, the corresponding entry is automatically removed from the WeakHashMap.

**This means the garbage collector dominates the weakHashMap. (**It is very useful when you have a large amount of objects, and you wish for them to be garbage collected when they are no longer in use, even if they are in the map.**)**

**SortedMap Interface in Java:**

**----------------------------------------**

**Definition:** The SortedMap interface is a sub-interface of the Map interface. It represents a map that maintains its entries in ascending order **based on the keys.** The order is determined either by the natural ordering of the keys or by a specified comparator.

**Purpose:** It is used when you need a map with keys sorted in a specific order. It is particularly useful for range views (e.g., retrieving a subset of entries).

Key Characteristics:

---------------------------

**Sorted Keys:** Entries are sorted by keys in ascending order.

**No Duplicate Keys:** Like all maps, it does not allow duplicate keys.

**Allows null:** Depending on the implementation, it may or may not allow null keys or values.

The SortedMap interface extends the Map interface and adds the following methods:

**Accessing Elements: Comparator<? super K> comparator():** Returns the comparator used to order the keys, or null if natural ordering is used.

**K firstKey():** Returns the first (lowest) key in the map.

**K lastKey():** Returns the last (highest) key in the map.

**Range Views: SortedMap<K, V> headMap(K toKey):** Returns a view of the portion of the map whose keys are less than toKey.

**SortedMap<K, V> tailMap(K fromKey):** Returns a view of the portion of the map whose keys are greater than or equal to fromKey.

**SortedMap<K, V> subMap(K fromKey, K toKey):** Returns a view of the portion of the map whose keys range from fromKey (inclusive) to toKey (exclusive).

**Inherited Methods:** Inherits all methods from the Map interface (e.g., put(), get(), remove(), size()).

The SortedMap interface is implemented by the following classes:

**TreeMap:**

**--------------**

TreeMap is a class in Java that implements the SortedMap and NavigableMap interfaces. It stores data in key-value pairs and maintains elements in sorted order based on the keys. It is part of the java.util package.

**Purpose:** It is used to store and retrieve data in key-value pairs while maintaining a sorted order of keys. It is particularly useful for range queries and operations that require sorted data.

Key Characteristics:

---------------------------

**Sorted Order:** Elements are sorted by keys in ascending order (natural order or by a comparator).

**Red-Black Tree:** It uses a Red-Black Tree (a self-balancing binary search tree) for storage.

**No Duplicate Keys:** Like all maps, it does not allow duplicate keys.

**No null Keys:** It does not allow null keys (if natural ordering is used).

**Allows null Values:** It allows null values.

**Not Synchronized:** It is not thread-safe by default.

How TreeMap Works: TreeMap uses a Red-Black Tree to store key-value pairs. A Red-Black Tree is a self-balancing binary search tree that ensures operations like insertion, deletion, and search take O(log n) time.

**Sorting:** By default, keys are sorted in natural order (e.g., alphabetical order for strings, ascending order for numbers). You can also provide a custom Comparator to define a specific sorting order.

**Performance:** Provides O(log n) time complexity for basic operations like get, put, and remove.

Constructors of TreeMap:

-----------------------------------

**TreeMap():** Creates an empty TreeMap sorted according to the natural ordering of its keys.

Example: TreeMap<String, Integer> map = new TreeMap<>();

**TreeMap(Comparator<? super K> comparator):** Creates an empty TreeMap sorted according to the specified comparator.

Example: TreeMap<String, Integer> map = new TreeMap<>(Comparator.reverseOrder());

**TreeMap(Map<? extends K, ? extends V> m):** Creates a TreeMap containing the key-value pairs of the specified map, sorted according to natural ordering.

Example: Map<String, Integer> oldMap = new HashMap<>();

oldMap.put("Apple", 1);

TreeMap<String, Integer> map = new TreeMap<>(oldMap);

**TreeMap(SortedMap<K, ? extends V> m):** Creates a TreeMap containing the key-value pairs of the specified sorted map, using the same ordering.

Example: SortedMap<String, Integer> sortedMap = new TreeMap<>();

sortedMap.put("Banana", 2);

TreeMap<String, Integer> map = new TreeMap<>(sortedMap);

**→** In TreeMap, when relying on the default natural sorting order, **the keys must be homogeneous (of the same type or mutually comparable types)** and implement the Comparable interface. Otherwise, a **ClassCastException** (CCE) will be thrown at runtime.

**→** If you're providing a custom Comparator, the keys don't need to implement Comparable, but your Comparator must be able to handle them. There are no restrictions on the types of values you can store in a TreeMap. Values can be heterogeneous and don't need to implement Comparable. The values are not used for sorting, so there is no need for them to be comparable.

(Therefore, you can indeed store heterogeneous, non-comparable objects as values in a TreeMap, regardless of whether you're using default or customized sorting for the keys.)

**Example - 1:** TreeMap<Integer, String> map = new TreeMap<>();

map.put(5, "Java");

map.put(3, "C++");

map.put(1, "React");

// map.put(null, "Angular"); **//** **java.lang.NullPointerException**

System.out.println(map); **// {1=React, 3=C++, 5=Java}**

**Example - 2:**

import java.util.Comparator;

import java.util.TreeMap;

public class Test {

public static void main(String[] args) {

TreeMap<Integer, String> map = new TreeMap<>(new MyComparator());

map.put(5, "Java");

map.put(3, "C++");

map.put(1, "React");

System.out.println(map); **// {5=Java, 3=C++, 1=React}**

}

}

class MyComparator implements Comparator<Integer> {

@Override

public int compare(Integer i1, Integer i2) {

return i2.compareTo(i1);

}

}

**Hashtable in Java:**

**-------------------------**

**Definition:** Hashtable is a legacy class in Java that implements the Map interface. It stores data in key-value pairs and uses a hash table for storage. It is part of the java.util package and has been present since Java 1.0.

**Purpose:** It is used to store and retrieve data efficiently based on keys. It provides thread-safe operations, making it suitable for multi-threaded environments.

Key Characteristics:

--------------------------

**Synchronized:** All methods are synchronized, making it thread-safe.

**No null Keys or Values:** Does not allow null keys or null values.

**Initial Capacity:** Default initial capacity is 11 with a load factor of 0.75.

**Legacy Class:** While still functional, newer alternatives like HashMap are preferred for non-threaded scenarios.

How Hashtable Works:

-------------------------------

**Hashing Mechanism:**  Uses a hash function to compute an index (hash code) for each key. This index determines where the key-value pair is stored in the internal array.

**Collision Handling:** If two keys have the same hash code (collision), they are stored in the same bucket as a linked list.

**Load Factor:** The load factor (default 0.75) determines when the table is rehashed (increased in size). When the number of entries exceeds capacity \* loadFactor, the table is rehashed.

**→** While you can technically add heterogeneous objects as keys in Hashtable (because it stores Object references), it's generally considered very bad practice. Hashtable relies on the **hashCode() and equals()** methods of the key objects.

Hashtable implements the **Serializable and Cloneable interfaces.** Hashtable does not implement the RandomAccess interface. Hashtable is suitable for search operations due to its hash table implementation.

Constructors of Hashtable:

------------------------------------

**Hashtable():** Creates an empty Hashtable with default initial capacity **(11) and load factor (0.75).**

**Ex:** Hashtable<String, Integer> ht = new Hashtable<>();

**Hashtable(int initialCapacity):** Creates an empty Hashtable with specified initial capacity and default load factor (0.75). **Ex:** Hashtable<String, Integer> ht = new Hashtable<>(20);

**Hashtable(int initialCapacity, float loadFactor):** Creates an empty Hashtable with specified initial capacity and load factor.  **Ex:** Hashtable<String, Integer> ht = new Hashtable<>(20, 0.85f);

**Hashtable(Map<? extends K, ? extends V> t):** Creates a Hashtable initialized with the key-value pairs from the specified map. **Example:** Map<String, Integer> map = new HashMap<>();

map.put("A", 1);

Hashtable<String, Integer> ht = new Hashtable<>(map);

Example:

import java.util.Hashtable;

public class Test {

public static void main(String[] args) {

Hashtable<Temp, String> table = new Hashtable<>();

table.put(new Temp(5), "Five");

table.put(new Temp(2), "Two");

table.put(new Temp(6), "Six");

table.put(new Temp(15), "Fifteen");

table.put(new Temp(23), "Twenty three");

table.put(new Temp(16), "Sixteen");

System.out.println(table); **// {6=Six, 16=Sixteen, 5=Five, 15=Fifteen, 2=Two, 23=Twenty three}**

}

}

class Temp {

int i;

Temp(int i) {

this.i = i;

}

@Override

public int hashCode() {

return i;

}

@Override

public String toString() {

return i + "";

}

}

**(** **Implementation Variability:** The internal structure and traversal algorithms of Hashtable can vary across different Java Virtual Machine (JVM) implementations and even across different versions of the same JVM. This means that the **"top-to-bottom bucket, right-to-left within bucket"** behavior you're describing might be observed in some environments but not in others.

**Hashtable's Contract:** The Java documentation for Hashtable does not specify any particular order for retrieving elements. The contract of the Hashtable class is that it provides key-based access to values, not ordered retrieval.

**Hash Code Distribution:** The order in which buckets are accessed is heavily dependent on the hash codes of the keys. Even if there were a consistent internal traversal algorithm, the order of elements would still be unpredictable due to the nature of hash codes.

**Bucket Changes:** When the internal table of the hashtable is resized, the elements within the buckets will be redistributed, and the order will change.**)**

**→** Hardcoding frequently changing values (like usernames, passwords, email addresses, database connection strings, etc.) directly into Java code leads to:

**Recompilation:** Any change requires modifying the source code and recompiling the Java classes.

**Rebuilding:** The application needs to be rebuilt into a new deployable artifact (e.g., a WAR or JAR file).

**Redeployment:** The new artifact must be deployed to the application server.

**Server Restarts:** In some cases, a server restart might be necessary to pick up the changes.

**Business Impact:** These steps can cause downtime and disruption, leading to a significant business impact.

**Solution:**  Store configuration values in external properties files (e.g., config.properties). These files are plain text files with key-value pairs. Use the **java.util.Properties** class to read the properties file into your Java program. Access the values using the corresponding keys.

**→ The Properties class** is intended for storing and retrieving configuration data where **both keys and values are strings.** It provides methods to load and save properties to and from files (typically .properties files). The Properties class **extends Hashtable<Object, Object>**, but it's used with String keys and values.

**Methods of the Properties Class:**

**String getProperty(String key):** Retrieves the value associated with the specified key. Returns null if the key is not found.

**Object setProperty(String key, String value):** Sets the property associated with the specified key to the given value. Returns the **previous value** of the specified key in this property list, or **null** if it did not have one.

**Enumeration<?> propertyNames():** Returns an enumeration of all the keys in the property list, including the default properties.

**void load(InputStream inputStream):** Reads property key-value pairs from the specified InputStream. Used to load properties from a file or other input source.

**void store(OutputStream outputStream, String comments):** Writes property key-value pairs to the specified OutputStream. Used to save properties to a file or other output destination. The comments parameter is a description of the property list.

**Example:**

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Properties;

public class Test {

public static void main(String[] args) throws IOException {

Properties p1 = new Properties();

FileInputStream fis = new FileInputStream("Devi.properties");

p1.load(fis);

System.out.println(p1); **// {password=pswd1289, location=bhadrak, user=devi}**

System.out.println(p1.getProperty("user")); **// devi**

p1.setProperty("pin", "756114");

FileOutputStream fos = new FileOutputStream("Devi.properties");

p1.store(fos, "Updated by devi for demo code");

}

}

Devi.properties file:

---------------------------

#Updated by devi for demo code

#Mon Mar 24 21:35:20 IST 2025

location=bhadrak

password=pswd1289

pin=756114

user=devi

**Queue Interface in Java:**

**---------------------------------**

**Definition:** The Queue interface is part of the Java Collections Framework (java.util package). It represents a collection designed for holding elements prior to processing. It follows the **FIFO (First-In-First-Out)** principle by default, though some implementations may use different ordering.

**Purpose:** Used when elements need to be processed in a specific order (typically FIFO). Commonly used in scenarios like task scheduling, buffering, and breadth-first search algorithms.

**Key Characteristics:** Ordered collection (but not necessarily sorted). Allows duplicate elements. Typically does not allow null elements (implementation dependent). Supports operations at both ends (head for removal, tail for insertion).

The Queue interface provides three groups of methods:

Basic Operations:

------------------------

**boolean add(E e):** Inserts element (throws exception if capacity exceeded).

**boolean offer(E e):**  Inserts element (returns false if capacity exceeded).

**E remove():** Removes and returns head (throws exception if empty).

**E poll():** Removes and returns head (returns null if empty).

**E element():** Returns head without removal (throws exception if empty).

**E peek():** Returns head without removal (returns null if empty).

Queue Implementations:

----------------------------------

**LinkedList:** Implements both Queue and Deque interfaces. Allows null elements. Not thread-safe. Example:

Queue<String> queue = new LinkedList<>();

**PriorityQueue:** Elements ordered according to natural ordering or comparator. Does not allow null elements. Not thread-safe. Example: Queue<Integer> pq = new PriorityQueue<>();

**ArrayDeque:** Resizable-array implementation. More efficient than LinkedList for many operations. Not thread-safe.

Example: Queue<String> deque = new ArrayDeque<>();

**Blocking Queues (in java.util.concurrent):** ArrayBlockingQueue, LinkedBlockingQueue, etc. Thread-safe implementations with capacity restrictions. Support blocking operations.

**PriorityQueue in Java:**

**------------------------------**

**Definition:** PriorityQueue is a class that implements the Queue interface. It stores elements in a priority heap where elements are ordered according to their natural ordering or by a custom Comparator. It is part of the java.util package.

**Purpose:** Used when elements need to be processed based on priority rather than FIFO order. Commonly used in algorithms like Dijkstra's shortest path, Huffman coding, and task scheduling.

Key Characteristics:

--------------------------

**Unbounded Size:** Grows dynamically (though has a default initial capacity of **11**).

**No null Values:** Doesn't permit null elements.

**Not Thread-Safe:** Requires external synchronization for concurrent access.

**Non-Synchronized:** For thread-safe alternatives, use **PriorityBlockingQueue**.

**Natural Ordering (Default):** Elements are ordered in ascending order (for numbers) or alphabetical order (for strings).

Example: PriorityQueue<Integer> pq = new PriorityQueue<>();

**Custom Ordering (Using Comparator):** Elements can be ordered in descending order or any custom order. Example:

PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());

**Constructors of PriorityQueue:**

**------------------------------------------**

**PriorityQueue():** Creates a PriorityQueue with default initial capacity **(11) and natural ordering.** Example:

PriorityQueue<Integer> pq = new PriorityQueue<>();

**PriorityQueue(int initialCapacity):** Creates a PriorityQueue with specified initial capacity and natural ordering.

Example: PriorityQueue<Integer> pq = new PriorityQueue<>(20);

**PriorityQueue(Comparator<? super E> comparator):** Creates a PriorityQueue with default initial capacity (11) and custom

comparator. Example:

PriorityQueue<Integer> pq = new PriorityQueue<>(Comparator.reverseOrder());

**PriorityQueue(int initialCapacity, Comparator<? super E> comparator):** Creates a PriorityQueue with specified initial capacity and custom comparator. Example:

PriorityQueue<Integer> pq = new PriorityQueue<>(20, Comparator.reverseOrder());

**PriorityQueue(Collection<? extends E> c):** Creates a PriorityQueue containing elements from the specified collection.

**Example:** List<Integer> list = Arrays.asList(5, 3, 8);

PriorityQueue<Integer> pq = new PriorityQueue<>(list);

**PriorityQueue(SortedSet<? extends E> c):** Creates a PriorityQueue containing elements from a sorted set. Example:

SortedSet<Integer> set = new TreeSet<>(Arrays.asList(5, 3, 8));

PriorityQueue<Integer> pq = new PriorityQueue<>(set);

**( Key Points to Remember:** Elements are ordered based on priority (natural or custom ordering). Internally uses a priority heap (typically a binary heap).

**Time Complexity:**  Insertion (offer/add): O(log n), Removal (poll/remove): O(log n), Peek (peek/element): O(1)

**No null Values:** Attempting to add null throws NullPointerException.

**When to Use PriorityQueue:** Task scheduling with priorities. Any scenario where elements need to be processed based on priority rather than insertion order.**)**

**→ PriorityQueue does allow duplicate objects.** It does not enforce uniqueness like a Set. The priority queue orders elements based on their priority, but it doesn't prevent multiple elements with the same priority (or the same content) from being added.

If you rely on the default natural ordering of elements in a PriorityQueue, the objects must be **homogeneous and implement the Comparable interface** or else you will get a **ClassCastException (CCE)** at runtime when the PriorityQueue attempts to compare them.

**Comparator:**  If you provide a custom Comparator to the PriorityQueue's constructor, you can sort heterogeneous or non-comparable objects, as long as your Comparator can handle them.

**→** The behavior of **thread priorities and PriorityQueue** can be **platform-dependent**, leading to potential **inconsistencies** and limitations. Example:

import java.util.PriorityQueue;

import java.util.Comparator;

public class Test {

public static void main(String[] args) {

PriorityQueue<String> pq = new PriorityQueue<>(new MyComparator());

pq.offer("A");

pq.offer("B");

pq.offer("C");

pq.offer("D");

System.out.println(pq); **// [D, C, B, A]**

System.out.println(pq.poll()); **//**  **D**

}

}

class MyComparator implements Comparator<String> {

@Override

public int compare(String s1, String s2) {

return s2.compareTo(s1);

}

}

**NavigableSet and NavigableMap (Java 1.6 Enhancements):**

**--------------------------------------------------------------------------------**

Introduced in Java 1.6, NavigableSet and NavigableMap are enhanced interfaces that extend SortedSet and SortedMap, respectively. They provide navigation methods for efficiently accessing elements based on their position in the sorted collection.

NavigableSet:

-------------------

**Definition:**  A NavigableSet is a SortedSet with additional methods for closest-match navigation (e.g., finding elements just above/below a given value).

**Key Features:** Extends SortedSet. Provides methods to find elements relative to a given value. Implemented by TreeSet.

Methods in NavigableSet:

-----------------------------------

**E lower(E e):** Returns the greatest element < e (or null if none).

**E floor(E e):** Returns the greatest element ≤ e (or null).

**E ceiling(E e):** Returns the smallest element ≥ e (or null).

**E higher(E e):** Returns the smallest element > e (or null).

**E pollFirst():** Removes and returns the first (lowest) element.

**E pollLast():** Removes and returns the last (highest) element.

**NavigableSet<E> descendingSet():** Returns a reverse-ordered view of the set.

Example:

NavigableSet<Integer> set = new TreeSet<>(Arrays.asList(2, 4, 6, 8));

System.out.println(set.lower(5)); // 4 (greatest < 5)

System.out.println(set.higher(5)); // 6 (smallest > 5)

System.out.println(set.pollFirst()); // 2 (removes and returns first element)

**NavigableMap:**

**---------------------**

**Definition:** A NavigableMap is a SortedMap with navigation methods for keys, similar to NavigableSet.

**Key Features:** Extends SortedMap. Provides methods to find entries relative to a given key. Implemented by TreeMap.

**Methods in NavigableMap:**

**-------------------------------------**

Map.Entry<K,V> lowerEntry(K key): Returns the entry with the greatest key < key.

Map.Entry<K,V> floorEntry(K key): Returns the entry with the greatest key ≤ key.

Map.Entry<K,V> ceilingEntry(K key): Returns the entry with the smallest key ≥ key.

Map.Entry<K,V> higherEntry(K key): Returns the entry with the smallest key > key.

K lowerKey(K key): Returns the greatest key < key (no entry needed).

K floorKey(K key): Returns the greatest key ≤ key.

K ceilingKey(K key): Returns the least key ≥ key.

K higherKey(K key): Returns the least key > key.

K pollFirstEntry(): Removes and returns the first (lowest) entry.

K pollLastEntry(): Removes and returns the last (highest) entry.

NavigableMap<K,V> descendingMap(): Returns a reverse-ordered view of the map.

Example:

NavigableMap<String, Integer> map = new TreeMap<>();

map.put("A", 1);

map.put("C", 3);

map.put("B", 2);

System.out.println(map.lowerKey("C")); // "B" (greatest key < "C")

System.out.println(map.higherEntry("A")); // "B=2" (smallest key > "A")

System.out.println(map.pollFirstEntry()); // "A=1" (removes first entry)

Why Were They Added in Java 1.6?

-----------------------------------------------

**Enhanced Navigation:** Before Java 1.6, SortedSet and SortedMap only allowed basic range views (headSet, tailSet). NavigableSet/NavigableMap added fine-grained navigation (e.g., "find the closest key").

**Efficient Operations:** Methods like pollFirst() and pollLast() enable O(log n) removal of extremal elements (useful for priority-based algorithms).

**Reverse-Order Views:** descendingSet() and descendingMap() provide lazy-reversed views without copying data.

K**ey Differences from SortedSet/SortedMap:**

**Feature SortedSet/SortedMap NavigableSet/NavigableMap**

**-----------------------------------------------------------------------------------------------------------------------------------------------------------**

**Navigation** Only basic range views (subSet, headSet) Advanced closest-match methods (lower,

floor, etc.)

**Extremal Access** No direct removal methods pollFirst(), pollLast()

**Reverse View** Not supported descendingSet(), descendingMap()

**Collections Class:**

**------------------------**

The java.util.Collections class provides static utility methods for working with collections (e.g., List, Set, Map). It helps in performing operations like sorting, searching, synchronization, and more.

Key Methods in Collections:

-------------------------------------

**sort(List<T> list):** Sorts a list in natural order.

**sort(List<T> list, Comparator<T> c):** Sorts a list using a custom comparator.

**reverse(List<?> list):** Reverses the order of elements in a list.

**shuffle(List<?> list):** Randomly shuffles elements.

**Searching:**

**--------------**

**binarySearch(List<? extends Comparable<? super T>> list, T key):** Searches a sorted list using binary search.

**binarySearch(List<? extends T> list, T key, Comparator<? super T> c):** Searches with a custom comparator.

**Synchronization:**

**-----------------------**

**synchronizedList(List<T> list):** Returns a thread-safe synchronized list.

**synchronizedSet(Set<T> s):** Returns a thread-safe synchronized set.

**synchronizedMap(Map<K,V> m):** Returns a thread-safe synchronized map.

**Immutable Collections:**

**---------------------------------**

**unmodifiableList(List<? extends T> list):** Returns an immutable (read-only) list.

**unmodifiableSet(Set<? extends T> s):** Returns an immutable set.

**unmodifiableMap(Map<? extends K, ? extends V> m):** Returns an immutable map.

**Other Utilities:**

**---------------------**

**min(Collection<? extends T> coll):** Finds the minimum element (natural order).

**max(Collection<? extends T> coll):** Finds the maximum element (natural order).

**frequency(Collection<?> c, Object o):** Counts occurrences of an element.

**replaceAll(List<T> list, T oldVal, T newVal):** Replaces all occurrences of a value.

Example Usage:

---------------------

List<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2, 8, 1));

Collections.sort(numbers); // [1, 2, 5, 8]

Collections.reverse(numbers); // [8, 5, 2, 1]

Collections.shuffle(numbers); // Random order

int min = Collections.min(numbers); // 1

int max = Collections.max(numbers); // 8

List<Integer> syncList = Collections.synchronizedList(numbers); // Thread-safe

List<Integer> unmodifiableList = Collections.unmodifiableList(numbers); // Read-only

**Arrays Class:**

**------------------**

The java.util.Arrays class provides static **utility methods** for working with arrays (e.g., sorting, searching, converting to collections).

**Key Methods in Arrays:**

**--------------------------------**

Sorting & Searching: Supported Types

---------------------------- --------------------------------

**sort(array):** Sorts an array in ascending order. (All primitive & Object[])

**sort(T[] a, Comparator<? super T> c):** Sorts an array using a custom comparator. (Object[])

**binarySearch(array, key):** Searches a sorted array using binary search. (All primitive & Object[])

**binarySearch(T[] a, T key, Comparator<? super T> c):** Searches with a custom comparator. (Object[])

Comparison & Copying:

--------------------------------

**equals(array1, array2):** Checks if two arrays are equal. (All primitive & Object[])

**copyOf(original array, newLength):** Copies an array with a new length. (All primitive & Object[])

Filling & Conversion:

----------------------------

**fill(array, value):** Fills an array with a specific value. (All primitive & Object[])

**asList(T... a):** Converts an array to a fixed-size List. (Object[] (varargs))

**toString(array):** Returns a string representation of an array. (All primitive & Object[])

Stream Operations (Java 8+):

---------------------------------------

**stream(array):** Converts an array to a Stream. (All primitive & Object[])

**parallelSort(array):** Sorts an array in parallel (faster for large arrays). (All primitive & Object[])

**Example Usage:**

**----------------------**

int[] numbers = {5, 2, 8, 1};

Arrays.sort(numbers); // [1, 2, 5, 8]

int index = Arrays.binarySearch(numbers, 5); // 2

int[] copy = Arrays.copyOf(numbers, numbers.length \* 2); // [1, 2, 5, 8, 0, 0, 0, 0]

Arrays.fill(copy, 0); // [0, 0, 0, 0, 0, 0, 0, 0]

List<Integer> list = Arrays.asList(1, 2, 3); // Fixed-size list

String arrayStr = Arrays.toString(numbers); // "[1, 2, 5, 8]"

Key Differences Between Collections and Arrays:

**Feature Collections Arrays**

**------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Works with** List, Set, Map Primitive & object arrays

**Sorting** Collections.sort(list) Arrays.sort(array)

**Binary Search** Collections.binarySearch(list, key) Arrays.binarySearch(array, key)

**Thread Safety** synchronizedList(), synchronizedSet() Not applicable

**Immutability** unmodifiableList(), unmodifiableSet() Not applicable

**Conversion** Collections.addAll(list, elements) **Arrays.asList(array)**

**(** Use Collections when working with **List, Set, or Map**. Use Arrays when working with **primitive or object arrays. Arrays.asList()** is useful for converting arrays to lists (but the list is fixed-size). Both classes are essential for efficient data manipulation in Java.**)**

**→** If a List contains heterogeneous objects that cannot be compared with each other (if Comparator is not provided), Collections.sort() will throw a **ClassCastException (CCE).** If a List contains null elements and Collections.sort() attempts to compare them, it will throw a **NullPointerException (NPE).** This is because null cannot be compared using compareTo() or with a comparator.

Example - 1:

-----------------

import java.util.ArrayList;

import java.util.Comparator;

import java.util.Collections;

public class Test {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<>();

list.add("D");

list.add("A");

list.add("Z");

list.add("K");

System.out.println(list); **// [D, A, Z, K]**

Collections.sort(list); **// [A, D, K, Z]**

System.out.println(list);

Collections.sort(list, new MyComparator());

System.out.println(list); **// [Z, K, D, A]**

}

}

class MyComparator implements Comparator<String> {

@Override

public int compare(String s1, String s2) {

return s2.compareTo(s1);

}

}

**Collections.binarySearch():** This method performs a binary search on a sorted List to find a specified element. It relies on the elements being sorted in ascending order. It can use either the natural ordering of the elements (if they implement Comparable) or a custom Comparator.

**Successful Search:** If the target element is found in the list, binarySearch() returns the index of the element.

**Unsuccessful Search:** If the target element is not found, binarySearch() returns a negative value. The negative value is calculated as **-(insertion point) - 1.** The "insertion point" is the index where the target element would be inserted to maintain the sorted order of the list.

**Example:** If binarySearch() returns -3, the insertion point is 2.

**→** Before calling Collections.binarySearch(), **the List must be sorted in ascending order.** If the List is not sorted, the results of binarySearch() are unpredictable and potentially incorrect.

**Comparator Consistency:** If the List was sorted using a custom Comparator, you must provide the same Comparator object

to binarySearch() during the search operation. Using a different Comparator or relying on natural ordering when the list

was sorted with a Comparator will lead to unpredictable and incorrect results.

**Unpredictable Results:** Unpredictable results can include incorrect indexes or incorrect negative return values. The binarySearch method **will not throw an exception** if the list is not sorted, it will simply return the incorrect result.

Example:

import java.util.ArrayList;

import java.util.Comparator;

import java.util.Collections;

public class Test {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<>();

list.add(3);

list.add(1);

list.add(8);

list.add(5);

Collections.sort(list, new MyComparator());

System.out.println(list); **// [8, 5, 3, 1]**

System.out.println(Collections.binarySearch(list, 5, new MyComparator())); **// 1**

System.out.println(Collections.binarySearch(list, 2, new MyComparator())); **// -4**

System.out.println(Collections.binarySearch(list, 3)); **// -1 (Unpredictable)**

}

}

class MyComparator implements Comparator<Integer> {

@Override

public int compare(Integer i1, Integer i2) {

return i2.compareTo(i1);

}

}

**→** The range of possible successful search results is from **0 to n - 1 (inclusive)**. The range of possible unsuccessful search results is from **-(n + 1) to -1 (inclusive)**. The overall range of possible return values for **Collections.binarySearch() is from -(n + 1) to n - 1 (inclusive).**

**Collections.reverse(List<?> list):** This method directly **modifies the order of elements** within the given List. It reverses the order of the elements in place. It is used to change the order of a list.

**Collections.reverseOrder():** This method **returns a Comparator** that imposes the **reverse of the natural ordering** on a collection of objects that implement the Comparable interface.

**Collections.reverseOrder(Comparator<T> cmp):** This method **returns a Comparator that reverses the order of the specified comparator.** It does not modify the original collection. It is used to create a Comparator for sorting in reverse order.

**(** The rules and behavior of Collections.binarySearch() and Arrays.binarySearch() are designed to be consistent. Both methods implement the standard binary search algorithm. The result ranges for successful and unsuccessful searches are the same for both methods. Both methods support using a custom Comparator to define the sorting order. Both methods require the data to be sorted before the search. Unsorted data will lead to unpredictable and incorrect results.

Collections.binarySearch() operates on List objects. **Arrays.binarySearch() operates on arrays (primitive or object arrays).** The underlying binary search algorithm and the rules for its application are the same for both. The only difference is the type of data structure they operate on.**)**

**→ Arrays.asList() does not create a new, independent List object.** Instead, it returns a fixed-size List view of the original array. This means that the List is backed by the array, and changes to one affect the other.

**Synchronization Between Array and List:** Changes made to the array using the array reference are immediately reflected in

the List view. Similarly, changes made to the List view (using methods like set()) are immediately reflected in the original

array.

**Fixed Size and Unsupported Operations:** The List view returned by Arrays.asList() has a fixed size, determined by the original array's length. Therefore, operations that would change the size of the List (e.g., add(), remove(), clear()) are not supported. Attempting to perform these operations will result in an **UnsupportedOperationException** at runtime.

**Array Type Safety and ArrayStoreException:** The List view maintains the type safety of the original array. Attempting to replace an element in the List with an object of a different type than the array's component type will result in an **ArrayStoreException** at runtime. This exception is thrown because the underlying array cannot store an object of that type.

**( Key takeaway:** Arrays.asList() provides a convenient way to view an array as a List, but it's crucial to remember that it's a view, not a copy, and it has size limitations.**) Example:**

import java.util.Arrays;

import java.util.List;

public class Test {

public static void main(String[] args) {

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

List<Integer> list = Arrays.asList(arr);

arr[0] = 10;

System.out.println(list); **// [10, 2, 3, 4]**

list.set(3, 20);

for (Integer i : arr) {

System.out.print(i+" "); **// 10 2 3 20**

}

}

}

**Concurrent Collection Classes in Java:**

**---------------------------------------------------**

Concurrent collections are thread-safe implementations of Java's collection framework designed for multi-threaded environments. They allow safe access and modification by multiple threads without explicit synchronization.

Why Use Concurrent Collections?

-----------------------------------------------

**Thread Safety:** Regular collections (e.g., ArrayList, HashMap) are not thread-safe. Concurrent collections (e.g., ConcurrentHashMap, CopyOnWriteArrayList) handle synchronization internally.

**Better Performance:** Traditional synchronization (synchronizedList, synchronizedMap) uses global locks, causing bottlenecks. Concurrent collections use fine-grained locking or lock-free algorithms for higher throughput.

**Avoid ConcurrentModificationException:** Regular collections throw this if modified while iterating. Concurrent collections allow safe iteration even during modifications. Example:

import java.util.ArrayList;

import java.util.Iterator;

public class Test {

public static void main(String[] args) throws InterruptedException {

MyThread.list.add(1);

MyThread.list.add(2);

MyThread.list.add(3);

MyThread.list.add(4);

MyThread mt = new MyThread();

mt.start();

Iterator<Integer> itr = MyThread.list.iterator();

while (itr.hasNext()) {

int i = (Integer) itr.next();

System.out.print(i + " ");

Thread.sleep(2000);

}

}

}

class MyThread extends Thread {

static ArrayList<Integer> list = new ArrayList<>();

@Override

public void run() {

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

list.add(10);

}

}

**Output:**  1 Exception in thread "main" **java.util.ConcurrentModificationException**

**Scalability:** Optimized for high-concurrency scenarios (e.g., web servers, distributed systems).

Key Concurrent Collection Classes:

**Class Interface Description When to Use**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**ConcurrentHashMap** ConcurrentMap Thread-safe HashMap with High-read, high-write scenarios (e.g.,

segment-level locking. caching).

**CopyOnWriteArrayList** List Thread-safe ArrayList where Frequent reads, rare writes (e.g.,

writes create a new copy. event listeners).

**CopyOnWriteArraySet** Set Thread-safe Set backed by Similar to CopyOnWriteArrayList

CopyOnWriteArrayList. but for sets.

**ConcurrentSkipListMap** ConcurrentNavigableMap Thread-safe TreeMap with Sorted maps with concurrent

log(n) operations. access.

**ConcurrentSkipListSet** ConcurrentNavigableSet Thread-safe TreeSet with Sorted sets with concurrent

log(n) operations. access.

**BlockingQueue**  BlockingQueue Thread-safe queue with blocking Producer-consumer patterns,

(e.g., ArrayBlockingQueue) operations (take(), put()). task scheduling.

**ConcurrentLinkedQueue** Queue Lock-free FIFO queue. High-throughput, non-blocking

scenarios.

**(** Use concurrent collections when: Multiple threads access/modify data. You need high performance without manual synchronization. You want to avoid ConcurrentModificationException. Concurrent collections make multi-threaded programming safer and more efficient! **)**

**ConcurrentMap Interface:** The ConcurrentMap is a thread-safe subinterface of Map that provides atomic operations for multi-threaded environments.

**Key Features:** Extends Map with atomic operations. Guarantees thread-safe put/get/remove operations. Implemented by ConcurrentHashMap.

**Important Methods:**

**Method Description**

**V putIfAbsent(K key, V value)** Adds key-value only if key is absent

**boolean remove(Object key, Object value)** Removes only if key-value pair matches

**V replace(K key, V value)** Replaces only if key exists

**boolean replace(K key, V oldVal, V newVal)**  Replaces only if key has oldVal

**Example:**

**-------------**

ConcurrentMap<String, Integer> map = new ConcurrentHashMap<>();

map.putIfAbsent("A", 1); // Only adds if "A" doesn't exist

map.replace("A", 1, 2); // Replaces 1 with 2 only if "A=1" exists

**ConcurrentHashMap:**

**-----------------------------**

ConcurrentHashMap is a thread-safe, high-performance implementation of the Map interface introduced in Java 5 as part of the **java.util.concurrent package.** It is designed for multi-threaded environments where multiple threads can read and write concurrently without explicit synchronization.

**Problem with Hashtable & synchronizedMap:** They use global locks, leading to poor performance under high concurrency. Only one thread can access the map at a time, causing bottlenecks.

**Solution:** ConcurrentHashMap uses **fine-grained locking** (segment-level locks in Java 7) and CAS (Compare-And-Swap in Java 8+) for better scalability.

Key Features:

------------------

**Thread Safety:** Supports concurrent reads & writes without full synchronization.

**High Performance:** Uses segment-level locks (Java 7) or CAS + synchronized blocks (Java 8+).

**No Null Keys/Values:** Throws NullPointerException if null is inserted.

**Fail-Safe Iterators:** Does not throw ConcurrentModificationException.

**Atomic Operations:** Provides methods like putIfAbsent(), compute(), merge().

Underlying Data Structure:

------------------------------------

**Java 7 (Segment-Based Locking):** Divided into 16 segments (default), each acting as a separate HashMap. Each segment has its own lock (reduces contention). **Reads are lock-free**, writes lock only the affected segment.

**Java 8+ (CAS + synchronized Blocks):** Replaced segments with Node-based buckets. Uses CAS (Compare-And-Swap) for lock-free reads. synchronized blocks for thread-safe writes. Tree bins (instead of linked lists) for collisions (if bucket size > 8).

**Constructors:**

**------------------**

**ConcurrentHashMap():** Default initial capacity (16), load factor (0.75), concurrency level (16).

**ConcurrentHashMap(int initialCapacity):** Custom initial capacity.

**ConcurrentHashMap(int initialCapacity, float loadFactor):** Custom initial capacity & load factor.

**ConcurrentHashMap(int initialCapacity, float loadFactor, int concurrencyLevel):** Custom initial capacity, load factor, and

concurrency level (Java 7).

**ConcurrentHashMap(Map<? extends K, ? extends V> m):** Creates from an existing map.

**Example:**

**-------------**

ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>(32, 0.8f);

Important Methods:

---------------------------

**V putIfAbsent(K key, V value):** Adds key-value only if key is absent.

**boolean remove(Object key, Object value):** Removes only if key-value pair matches.

**V replace(K key, V value):** Replaces only if key exists.

**boolean replace(K key, V oldVal, V newVal):** Replaces only if key has oldVal.

**Performance & Scalability:**

**Reads:** Lock-free (CAS in Java 8+).

**Writes:** Segment-level locking (Java 7) or synchronized on bucket (Java 8+).

**Concurrency Level:** Adjustable (default 16 in Java 7).

**Scalability:** Much better than Hashtable/synchronizedMap.

**Use ConcurrentHashMap when:** Multiple threads read/write concurrently. High throughput is needed (e.g., caching, shared data stores). Atomic operations (putIfAbsent, compute) are required.

**Summary:** Introduced in Java 5 as a high-concurrency alternative to Hashtable. Uses segment-level locks (Java 7) or CAS + synchronized (Java 8+). Supports atomic operations (putIfAbsent, compute, merge). Best for multi-threaded environments (e.g., caching, real-time systems). For high-performance concurrent programming, ConcurrentHashMap is the best choice!

**Example - 1:**

**-----------------**

ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

map.put(1, "A");

map.put(5, "Z");

map.putIfAbsent(3, "K");

map.putIfAbsent(1, "L");

map.remove(5, "Y");

map.replace(3, "K", "M");

System.out.println(map); **// {1=A, 3=M, 5=Z}**

**Example - 2:**

**-----------------**

import java.util.Iterator;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;

public class Test extends Thread {

static ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<>();

@Override

public void run() {

try {

Thread.sleep(2000);

} catch (InterruptedException e) {

e.printStackTrace();

}

map.put(3, "C");

}

public static void main(String[] args) throws InterruptedException {

map.put(1, "A");

map.put(5, "Z");

Test t1 = new Test();

t1.start();

Set<Integer> set = map.keySet();

Iterator<Integer> itr = set.iterator();

while (itr.hasNext()) {

Integer i = itr.next();

System.out.println("Current value of iterator : " + i + "....." + map.get(i));

Thread.sleep(2000);

}

System.out.println(map);

}

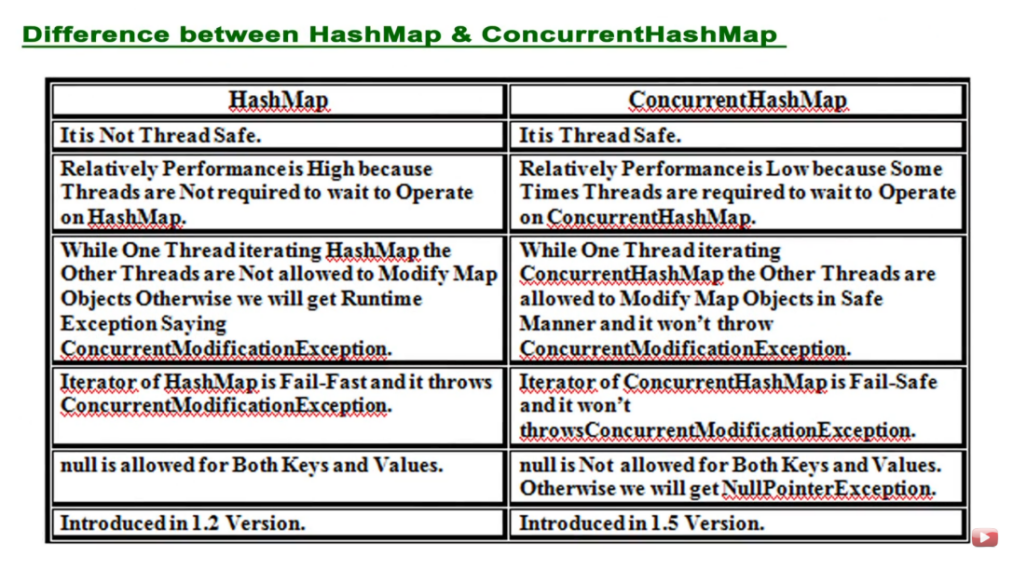
}

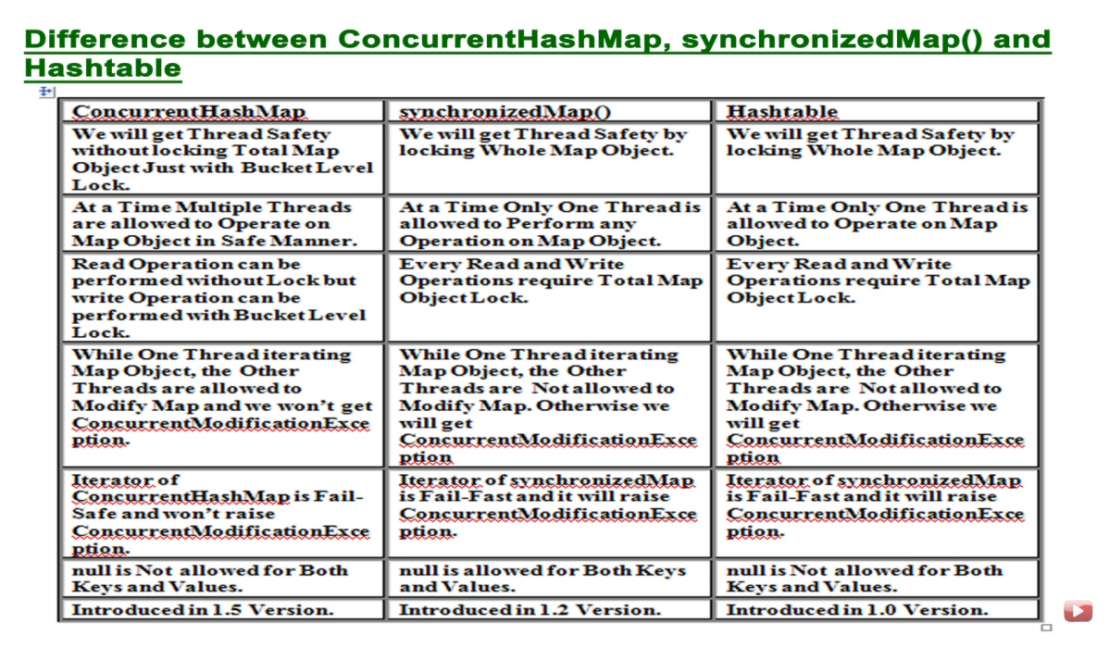
**Output:** Current value of iterator : 1.....A

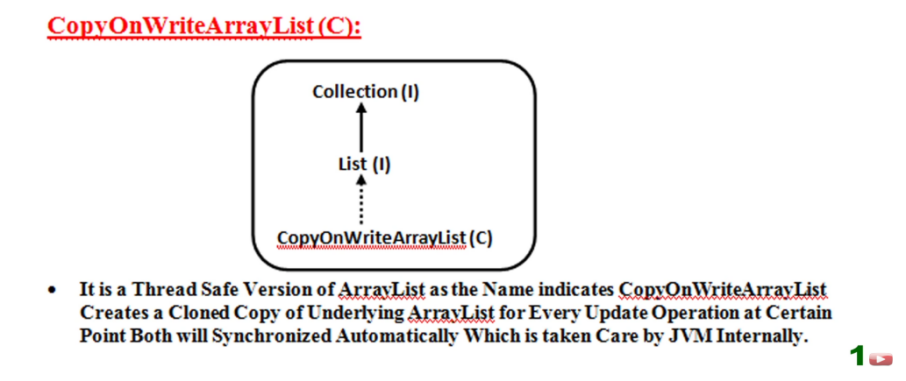
Current value of iterator : 5.....Z

{1=A, 3=C, 5=Z}

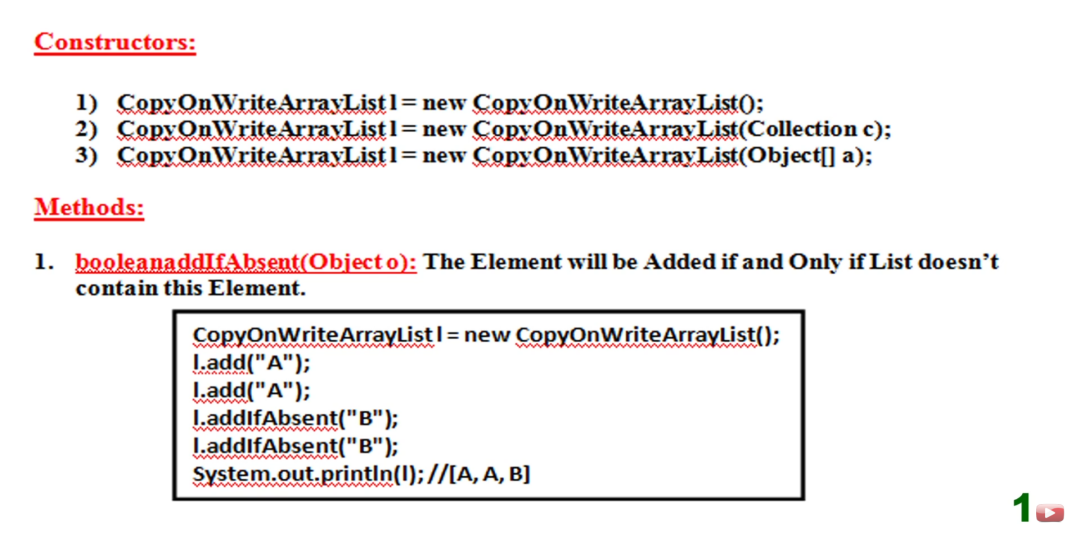
**→** ConcurrentHashMap iterators provide weak consistency, not strong consistency. They are designed for concurrent read operations and will not throw **ConcurrentModificationException.** The iterator **might or might not see changes made to the map** during iteration. The iterator is not guaranteed to see all changes. The output will depend on the timing of the threads and the order of the buckets within the map. **If the iterator has already passed the bucket where the new entry (3, "C") is inserted, it will not see that entry.** If the iterator has not yet reached that bucket, it might see the new entry.











**2. int addAllAbsent(Collection c):** The elements of the collection will be added to the list if elements are absent and returns the number of elements added.

ArrayList l = new ArrayList();

l.add("A");

l.add("B");

CopyOnWriteArrayList l1 = new CopyOnWriteArrayList();

l1.add("A");

l1.add("C");

System.out.println(l1);  **// [A, C]**

l1.addAll(l);

System.out.println(l1); **// [A, C, A, B]**

ArrayList l2 = new ArrayList();

l2.add("A");

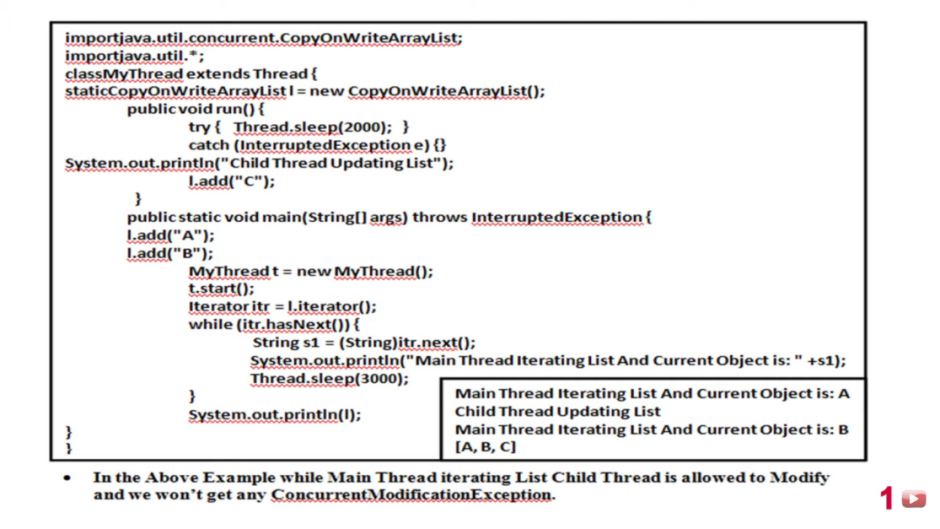
l2.add("D");

l1.addAllAbsent(l2);

System.out.println(l1); **// [A, C, A, B, D]**

**Example - 1:**





**→** If we replace CopyOnWriteArrayList with ArrayList, then we will get ConcurrentModificationException. **Iterator of CopyOnWriteArrayList can’t perform the remove operation.** Otherwise, we will get a RuntimeException: UnsupportedOperationException. **Example:**

import java.util.concurrent.CopyOnWriteArrayList;

import java.util.Iterator;

class Test {

public static void main(String[] args) {

CopyOnWriteArrayList l = new CopyOnWriteArrayList();

l.add("A");

l.add("B");

l.add("C");

l.add("D");

System.out.println(l); **// [A, B, C, D]**

Iterator itr = l.iterator();

while (itr.hasNext()) {

String s = (String) itr.next();

if (s.equals("D"))

itr.remove(); **// RE: java.lang.UnsupportedOperationException**

}

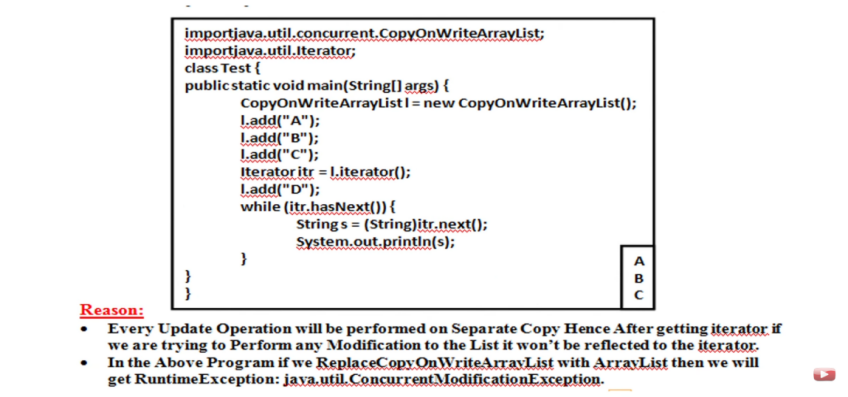
System.out.println(l);

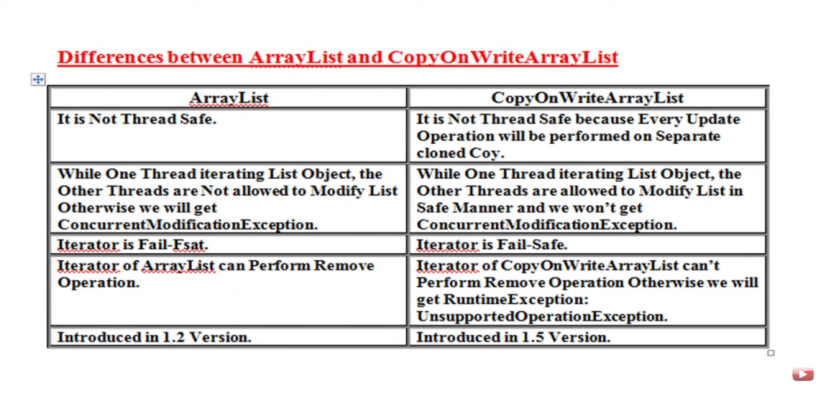
}

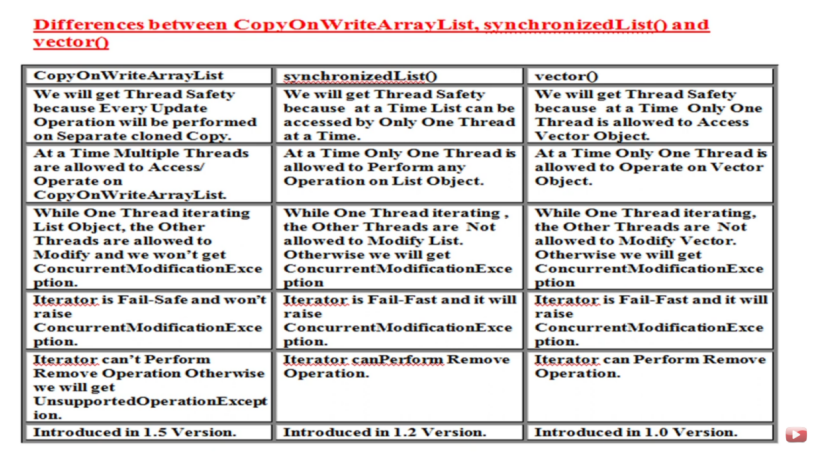
}

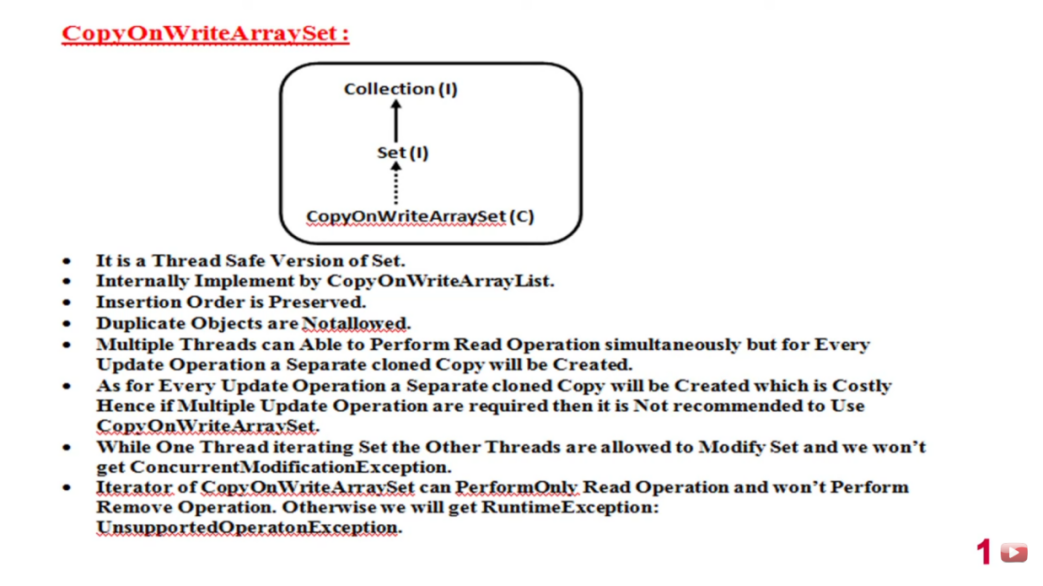
If we replace CopyOnWriteArrayList with ArrayList, we won’t get any UnsupportedOperationException. In this case, the output is: **[A, B, C, D]**

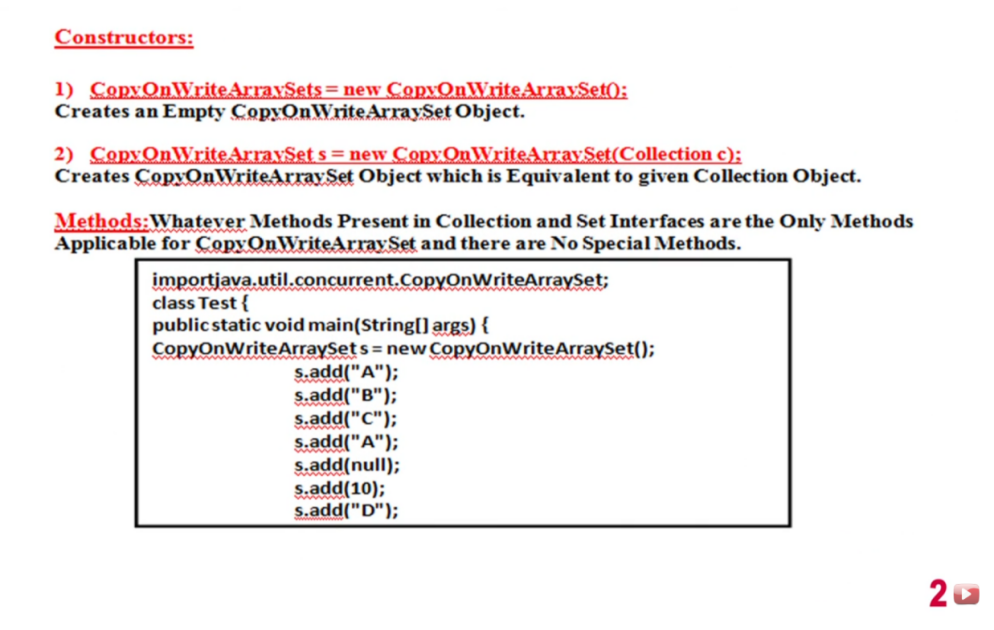
**[A, B, C]**

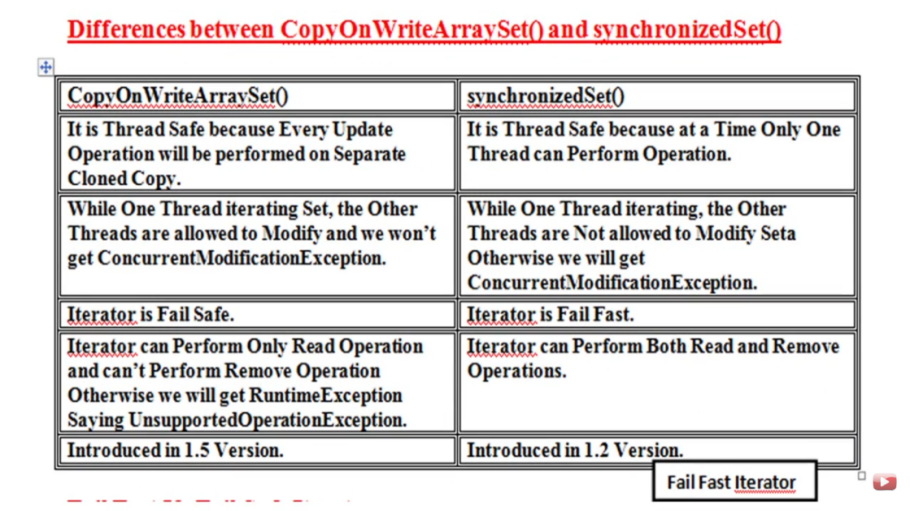












**Generics**

**====================================================================================**

**Main Objectives of Generics:**

**---------------------------------------**

**Type Safety:** Generics enable **compile-time type checking.** By parameterizing classes and methods with type parameters, the compiler can verify that you are using objects in a type-consistent way. This helps catch type-related errors early in the development process (at compile time) rather than at runtime, leading to more robust and reliable code.

**Resolution of Type-Casting Problems:** Without generics, when retrieving objects from collections (like ArrayList or HashMap), you would typically need to cast them to their specific type. This casting can lead to **ClassCastException** at runtime if the object's actual type is not what you expected.

Generics eliminate the need for most of these explicit casts. When you declare a collection with a specific type parameter (e.g., List<String>), the compiler knows the type of objects it will contain, and you can retrieve them directly without casting. This improves code readability and reduces the risk of runtime errors.

**(** In essence: Generics enhance the type system of Java, making code safer and easier to work with by shifting type checking from runtime to compile time and reducing the boilerplate of manual type casting. This leads to fewer bugs, improved code clarity, and better overall development efficiency.**)**

Arrays and Type Safety:

-------------------------------

**Homogeneous Data Types:** Arrays in Java are indeed type-safe. When you declare an array of a specific type (e.g., int[], String[], Object[]), the compiler enforces that you can only store elements of that declared type or its subtypes within the array.

**Compile-Time Error (CE):** If you attempt to store an element of an incompatible type into an array, the Java compiler will detect this type mismatch during compilation and issue a compile-time error. This prevents type-related errors from occurring at runtime.

Pre-Generics Collections and Lack of Type Safety:

------------------------------------------------------------------

**Heterogeneous Elements:** Before the introduction of generics in Java 1.5, collections like ArrayList, HashMap, HashSet, etc., were not inherently type-safe. They were designed to store objects of type Object. This meant you could add elements of any type to a collection without any compile-time errors.

**Runtime Type Casting and Potential Errors:**  To use the elements retrieved from these collections, you would typically **need to perform explicit type casting.** If the actual type of an element was not what you expected during the cast, it would result in a ClassCastException at runtime. This made code more prone to errors and harder to debug.

Example (Pre-Generics):

--------------------------------

import java.util.ArrayList;

import java.util.List;

public class NonGenericExample {

public static void main(String[] args) {

List list = new ArrayList();

list.add("Hello");

list.add(123); // No compile-time error

String str = (String) list.get(0); // Explicit cast - OK

System.out.println(str);

// String numStr = (String) list.get(1); // Explicit cast - Runtime error (ClassCastException)

// System.out.println(numStr);

}

}

**Generics to the Rescue:** It's important to note that generics were introduced in **Java 1.5** specifically to address the lack of **type safety in collections and the associated type-casting problems.** By using generics, you can parameterize collections with specific types, bringing compile-time type safety to collections as well. Example:

import java.util.ArrayList;

import java.util.List;

public class GenericExample {

public static void main(String[] args) {

List<String> stringList = new ArrayList<>();

stringList.add("Hello");

// stringList.add(123); // Compile-time error: Incompatible types

String str = stringList.get(0); // No explicit cast needed

System.out.println(str);

}

}

**→** Explicit type casting is generally not needed when retrieving elements from a generic collection or using generic methods. Because the compiler knows the type, when you retrieve an element from a generic collection (e.g., using list.get(index) from a List<String>), the compiler treats it directly as that type (String in this case). You don't need to explicitly cast it.

**ArrayList list = new ArrayList():**

**------------------------------------------**

**Raw type →** Doesn't specify the type of elements. Allows adding any type of objects (String, Integer, etc.).

**No compile-time type checking →** Type errors may occur at runtime. Not recommended after Java 5 (Generics introduction).

**ArrayList<String> list = new ArrayList<>():**

**--------------------------------------------------------**

**Generic type →** Specifies that the list will store only String elements.

**Type-safe →** Only String values can be added. Compile-time type checking, avoids ClassCastException at runtime. Recommended for modern Java coding.

**Polymorphism with Base Types:** The fundamental concept of polymorphism in Java allows a reference of a superclass (base type) to hold an object of its subclass (derived type). For example:

Object obj = new String("Hello"); // Valid: String is a subclass of Object

List list = new ArrayList(); // Valid (raw type, but illustrates the point)

**Generics and Type Parameters:** When you introduce generics, the type parameter becomes an integral part of the type itself. List<String> is a different type than List<Object>. **Even though String is a subclass of Object, List<String> is not considered a subtype of List<Object>.**

**Compiler Prevention:** To maintain type safety and prevent such runtime errors, the Java compiler strictly enforces the type parameter consistency during assignment.

**( In summary:** While polymorphism works beautifully with the base types of objects, it doesn't automatically extend to the parameter types of generic collections. The type parameter creates a specific contract for the collection, and the compiler ensures that this contract is upheld to guarantee type safety. You cannot directly assign a generic collection with a more specific type parameter to a reference with a more general type parameter.**)**

**→** When you define a generic class, interface, or method, the type parameters (the placeholders within <>) must be reference types (objects). This includes classes (both built-in and user-defined), interfaces, and even other generic types. **Primitive data types in Java cannot be directly used as type arguments for generics.**

If you attempt to use a primitive type as a type argument, the Java compiler will detect this and issue a compile-time error. Generics in Java are designed to work with objects, which are part of the Object hierarchy. Primitives are not part of this hierarchy; they are distinct, fundamental data types. Example:

ArrayList<int> list = new ArrayList<int>(); // Compile-time error

**Generic Version of Collection Classes (Java 1.5 onwards):** To achieve type-safety, Java introduced Generics in version 1.5. A collection class like ArrayList can be declared generically as:

class ArrayList<T> {

void add(T t);

T get(int index);

}

What this means: **T is a type parameter (placeholder for a data type).** When you create an ArrayList<String>, the T becomes String. This ensures that only String objects can be added, and retrieved safely.

**Generic Classes (or Template Classes):**

**----------------------------------------------------**

When you associate a type parameter (using angle brackets <T>, <E>, <K, V>, etc.) with a class definition, you are creating a generic class. These type parameters act as placeholders for the actual types that will be specified when an instance of the generic class is created.

The class's behavior and the types of its members (fields, methods) can then be defined in terms of these type parameters, making the class work with various types in a type-safe manner. The term "template classes" is often used in languages like C++, which have a similar concept to Java's generics.

**Defining Custom Generic Classes:** Java allows you to define your own generic classes to create reusable and type-safe components. This is incredibly powerful for creating data structures, algorithms, and utility classes that can work with different types without sacrificing type safety. **Example:**

public class Test {

public static void main(String[] args) {

Gen<String> ob1 = new Gen<String>("Java");

ob1.show(); // Type of object: java.lang.String

System.out.println(ob1.getObject()); // Java

Gen<Integer> ob2 = new Gen<Integer>(23); // Type of object: java.lang.Integer

ob2.show(); // 23

System.out.println(ob2.getObject());

}

}

class Gen<T> {

T object;

Gen(T object) {

this.object = object;

}

public void show() {

System.out.println("Type of object: " + object.getClass().getName());

}

T getObject() {

return object;

}

}

**Bounded Types in Generics:** In Java Generics, we can restrict the type parameter to a specific range using the **extends**

**keyword.** Such type parameters are called bounded types.

**Unbounded Type:** By default, if no bounds are specified, the type is unbounded, meaning any type (Object or subclass) can be passed. Example:

class Test<T> {

// T can be any type (unbounded)

}

**Bounded Type Syntax:** class Test<T extends X> { } Here, X can be either **a class or an interface.**

If X is a class: You can pass X or any subclass of X as the type parameter.

If X is an interface: You can pass X or any class that implements X. Example:

interface Runnable {}

class MyThread implements Runnable {}

class Test<T extends Runnable> { }

Test<Runnable> t1 = new Test<>();

Test<MyThread> t2 = new Test<>();

Test<String> t3 = new Test<>(); **// (Compile-time error)**

**Combining Bounds with &:** You can specify multiple bounds for a type parameter using the **& symbol. class Test<T extends Number & Runnable>** correctly declares that T must be a type that: Is a subclass of Number (or Number itself). Implements the Runnable interface. This allows you to enforce multiple type constraints on the generic type.

Why class Test<T extends Number & Thread> and class Test<T extends Runnable & Number> are Invalid:

-------------------------------------------------------------------------------------------------------------------------------------------

Class vs. Interface: In the combined bounds, at most one of the bounds can be a class. If you specify a class, **it must come first in the bounds list. All subsequent bounds must be interfaces.**

class Test<T extends Number & Thread> is invalid because Thread is a class and it appears after another class (Number). class Test<T extends Runnable & Number> is invalid because Number is a class and it appears after an interface (Runnable).

If you need a type that extends a class and implements multiple interfaces, the syntax is:

class Test<T extends SomeClass & InterfaceA & InterfaceB & ...> {

// ...

}

Where SomeClass is the class bound (at most one), and InterfaceA, InterfaceB, etc., are the interface bounds (zero or more).

**→** You **cannot directly use the implements keyword** for defining bounds. However the effect of requiring a type to implement an interface is achieved by using extends followed by the interface name (since extends is used for both inheritance from classes and implementation of interfaces in the context of generic bounds).

You also cannot use the super keyword for defining upper bounds. super is used for lower bounds in wildcard type arguments (e.g., List<? super Integer>), which is a different concept.

**Type Parameter Naming Convention:** Any valid Java identifier can be used as a type parameter name. However, there is a strong convention to use single uppercase letters to make them easily distinguishable from regular class or variable names. Common conventions include:

T: Type

E: Element (often used for collections)

K: Key (often used for maps)

V: Value (often used for maps)

N: Number

S, U, V etc.: Used for additional type parameters.

**Multiple Type Parameters:** You can indeed declare any number of type parameters for a generic class, interface, or method, separated by commas within the angle brackets <>. Each type parameter can have its own bounds (or no bounds). Example illustrating multiple bounded type parameters:

class Processor<T extends Number, U extends Runnable & Comparable<U>> {

private T value;

private U task;

public Processor(T value, U task) {

this.value = value;

this.task = task;

}

public T getValue() {

return value;

}

public void runTask() {

task.run();

}

// ... other methods

}

In this example, Processor is a generic class with two type parameters: T is bounded to be a subclass of Number. U is bounded to be a class that implements both the Runnable interface and the Comparable interface (and is comparable with itself).

**Generics with Wildcards in Java:**

**--------------------------------------------**

**m1(ArrayList<String> l):**  Allowed: Only ArrayList<String>.You can add only String type values. Compile-time error if you add anything other than String.

**m1(ArrayList<?> l):** Allowed: ArrayList of **any type** (e.g., ArrayList<Integer>, ArrayList<String>). You can't add anything

**except null**, because the exact type is unknown. Use case: **Read-only** operations.

**m1(ArrayList<? extends X> l):** X can be a class or interface. **If X is a class:** Accepts ArrayList<X> or its subclasses. **If X is an interface:** Accepts ArrayList<X> or its implementing classes. Can't add anything except null. Use case: Best for read-only operations.

**m1(ArrayList<? super X> l):** X can be a class or interface. **If X is a class:** Accepts ArrayList<X> or its superclasses. **If X is an interface:** Accepts ArrayList<X> or superclass of a class implementing X. Can add X type objects and null.

**Invalid Usage:** ArrayList<?> list = new ArrayList<? extends Number>(); // Compile-Time Error

**Reason:** Wildcards (?, ? extends, ? super) **are only allowed in declarations**, not in object creation.

**Correct way:** ArrayList<? extends Number> list;

list = new ArrayList<Integer>();

Type Parameter Declaration Levels:

-----------------------------------------------

**At Class Level:**

class Test<T> {

T obj;

void show() {

System.out.println(obj);

}

}

**At Method Level:** Must be declared just before return type. Example:

public <T> void m1(T obj) {

System.out.println(obj);

}

**Bounded Types at Method Level:** You can also define bounded type parameters at method level. Syntax:

public <T extends Runnable & Comparable<T>> void process(T obj) {

// logic here

}

You can extend one class and implement multiple interfaces using **‘&’**

**Generics and Non-Generic Method Interaction:** When a generic object is passed to a non-generic method, it behaves like a non-generic object (i.e., type-safety is lost). Similarly, when a non-generic object is passed to a generic method, it follows the generic constraints. Example Explained:

import java.util.ArrayList;

public class Test {

static void m1(ArrayList list) { // Non-generic method

list.add(true); // No compile-time error

list.add(10.5);

list.add(23);

}

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<String>(); // Generic list

list.add("Java");

list.add("C++");

m1(list); // Passing generic list to non-generic method

System.out.println(list); // Output: [Java, C++, true, 10.5, 23]

}

}

**Key Point:** The method m1(ArrayList list) doesn't enforce type-safety. Hence, even though list was declared as ArrayList<String>, the compiler allows addition of any type inside m1() because type information is erased at runtime due to type erasure.

Purpose of Generics:

----------------------------

**Type-safety:** Prevents adding incorrect types to collections.

**Avoids type casting:** No need to cast while retrieving elements.

**Compile-time feature:** All generic checks and constraints are applied at compile time.

**Type Erasure:** At runtime, all generic type information is removed (type erasure).

Hence, the JVM sees: ArrayList list = new ArrayList();

even if you wrote: ArrayList<String> list = new ArrayList<String>(); **Example - 1:**

ArrayList list = new ArrayList<String>();

list.add("Java");

list.add(10);

list.add(true);

System.out.println(list); // Output: [Java, 10, true]

No Compile-time Error (because we used raw type on LHS). No Runtime Error (because generic info is erased)

**Observation:** Since generics are erased, both:

ArrayList list = new ArrayList<String>();

ArrayList list = new ArrayList<Integer>(); are identical to JVM.

**Example - 2:**

import java.util.ArrayList;

public class Test {

void m1(ArrayList<String> list) { }

void m1(ArrayList<Integer> list) { }

public static void main(String[] args) { }

} // This code will result in compile-time error

**name clash:** m1(ArrayList<String>) and m1(ArrayList<Integer>) have the **same erasure.**

**Reason:** Generics use type erasure.

At compile time, both methods become: void m1(ArrayList list) { } and void m1(ArrayList list) { } ( Now both methods have the same signature, so the compiler throws an error due to method signature clash.)

**Behind the Scenes:** Java first compiles with generic info. Then applies type erasure (removes generic types). After erasure, both methods look identical. Java doesn't allow method overloading with same erased type, so error.

**Garbage Collection**

**====================================================================================**

In C++, the programmer is responsible for: **Creating new objects, Destroying useless objects**

However, programmers usually focus more on creating objects and neglect destroying them. Due to this negligence: Useless objects fill up memory. When trying to create new objects, memory may not be available. Leads to memory problems. Application may crash with **OutOfMemoryError.** Hence, out-of-memory is a very common problem in languages like C++.

Automatic Memory Management in Java: In Java, the programmer is responsible only for creating objects. Destroying useless objects is handled automatically by the **Garbage Collector (GC).** The primary objective of the garbage collector (GC) in Java is to automatically reclaim memory occupied by objects that are no longer in use by the application.

**The GC:** Is a **daemon thread** (runs in the background). Continuously monitors memory. Automatically destroys unused objects. As a result the chance of Java programs failing due to memory issues is very low. Java applications are more memory efficient compared to C++.

**→** Even though the programmer is not responsible for destroying objects in Java, it is highly recommended to make an object eligible for GC when it's no longer required. An object becomes eligible for GC if and only if no reference variable is pointing to it. In simple terms: "If no one is referring to the object anymore, it becomes garbage."

**Ways to Make an Object Eligible for Garbage Collection (GC) in Java:**

**---------------------------------------------------------------------------------------------**

In Java, objects are automatically garbage-collected when they are no longer reachable. Below are the 4 ways to make an object eligible for GC:

**1. Nullifying the Reference:** Set the reference variable to null so that the object becomes unreachable. Example:

Object obj = new Object(); // Object is reachable

obj = null; // Now eligible for GC

**2. Reassigning the Reference:** Assign the reference to another object, making the previous object unreachable. Ex:

Object obj1 = new Object(); // Object 1 is reachable

Object obj2 = new Object(); // Object 2 is reachable

obj1 = obj2; // Object 1 is now eligible for GC

**3. Local Variables (After Method Execution):** Objects created inside a method become eligible for GC once the method completes (unless returned or stored elsewhere). Example:

void createObject() {

Object obj = new Object(); // Eligible after method ends

}

**4. Island of Isolation (Cyclic References):** If two or more objects reference each other but are not reachable from any live reference, they become eligible for GC. Example:

class Person {

Person friend;

}

Person p1 = new Person();

Person p2 = new Person();

p1.friend = p2;

p2.friend = p1;

p1 = null;

p2 = null; // Both objects are now eligible for GC (Island of Isolation)

**How to Force GC** (Not Recommended): Java does not guarantee immediate garbage collection, but you can suggest GC using:  **System.gc();** // Hint to JVM (not guaranteed)

**Note:** Manually calling System.gc() is discouraged (JVM knows best when to run GC). GC eligibility does not mean immediate collection (depends on JVM's GC algorithm).

**Key Takeaways:** GC eligibility depends on reachability. The JVM decides when to actually collect objects. Use null, reassignment, or weak references to make objects unreachable. For memory-efficient programming, ensure unused objects are dereferenced properly!

**Note:** If an object has no reference variable, it is always eligible for Garbage Collection (GC). Even if an object has references, it can still be eligible for GC if all references are internal (i.e., no external reference exists). Example: **Island of Isolation**

Even though objects refer to each other, if they are not reachable from any live thread or root reference, they become eligible for GC.

class Node {

Node next;

}

public class Test {

public static void main(String[] args) {

Node a = new Node();

Node b = new Node();

Node c = new Node();

a.next = b;

b.next = c;

c.next = a;

a = null;

b = null;

c = null; // Now all 3 objects are referencing each other, but no external reference points to them.

// So they form an "island of isolation" — eligible for GC

}

}

Here are the two main ways to request the JVM to run the Garbage Collector (GC) in Java:

1. **Using System.gc():**

**----------------------------**

System.gc(); This is a **static method** of the System class. **Internally, it calls Runtime.getRuntime().gc();** It is just a request - JVM is not guaranteed to execute GC immediately.

1. **Using Runtime.getRuntime().gc():**

**-------------------------------------------------**

Runtime runtime = Runtime.getRuntime();

runtime.gc();

This explicitly accesses the Runtime instance and calls gc() on it. Also just a request - the JVM decides when to actually perform GC.

**Important Notes:** JVM may ignore these calls - it's not mandatory to run GC immediately. GC is handled by a low-priority daemon thread that runs in the background. These methods are hints to the JVM, not commands.

**Communication Between Java Application and JVM using Runtime Object:**

**-----------------------------------------------------------------------------------------------------**

The Runtime class is present in the java.lang package. It is a singleton class, which means only one instance exists per JVM.

To get the runtime object: Runtime r = Runtime.getRuntime();

Methods of Runtime Class:

------------------------------------

**totalMemory():**  Returns the total memory (in bytes) available in the JVM heap. Example: r.totalMemory();

**freeMemory():** Returns the free memory (in bytes) currently available in the heap. Example: r.freeMemory();

**gc():** Requests the JVM to run Garbage Collector. Example: r.gc();

**→** There are indeed **two gc() methods** in Java, one static in the System class and one instance method in the Runtime class, and they serve the same purpose: to suggest to the Java Virtual Machine (JVM) that it might be a good time to run the garbage collector.

**Conclusion:** Use Runtime.getRuntime().gc() when possible, as it gives more explicit control and is considered better practice even though both work similarly.

**Finalization and the finalize() Method in Java:**

**--------------------------------------------------------------**

**What is Finalization? :** Finalization is a mechanism in Java that allows an object to perform cleanup operations (like releasing system resources) just before it is garbage-collected. This is done using the finalize() method.

**The finalize() Method: Defined in the Object class** (can be overridden by any Java class). **Called by the Garbage Collector** (GC) before destroying an object. Syntax:

@Override

protected void finalize() throws Throwable {

// Cleanup code (e.g., closing files, releasing resources)

}

**How Finalization Works? :** When an object becomes unreachable, the JVM marks it for finalization. The Garbage Collector calls finalize() before reclaiming memory. After finalize() executes, the object is garbage-collected in the next cycle. Example:

class Resource {

@Override

protected void finalize() throws Throwable {

System.out.println("Finalize called! Cleaning up...");

super.finalize(); // Good practice to call parent's finalize()

}

}

public class Main {

public static void main(String[] args) {

Resource obj = new Resource();

obj = null; // Object becomes eligible for GC

System.gc(); // Suggests JVM to run GC (not guaranteed)

}

}

Possible Output: Finalize called! Cleaning up… (Note: finalize() may not always run immediately after GC is triggered.)

**Key Points About finalize():**

**Feature Details**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Invocation Guarantee** Not guaranteed (JVM decides).

**Execution Time** Non-deterministic (runs before GC, but timing is unpredictable).

**Performance Impact** Slows down GC (objects with finalize() require two GC cycles).

**Exception Handling** If finalize() throws an exception, it is ignored.

**Deprecation** Deprecated since Java 9 (use Cleaner or AutoCloseable instead).

**Why Was finalize() Deprecated? :**

**----------------------------------------------**

**Unpredictable Execution:** No guarantee when or if finalize() will run.

**Performance Overhead:** Objects with finalize() stay longer in memory (require extra GC cycles).

**Resource Leak Risk:** If finalize() fails, resources may never be released.

**Better Alternatives:** AutoCloseable (try-with-resources) for deterministic cleanup. java.lang.ref.Cleaner (Java 9+) for safer finalization.

**Best Alternative: AutoCloseable (Try-With-Resources):**

**--------------------------------------------------------------------------**

Instead of relying on finalize(), use:

class Resource implements AutoCloseable {

@Override

public void close() {

System.out.println("Resource closed!");

}

}

public class Main {

public static void main(String[] args) {

try (Resource res = new Resource()) {

// Use resource

} // Automatically calls close()

}

}

Output: Resource closed!

Advantages: Deterministic cleanup (runs immediately after try block). No GC dependency. No performance penalty.

**Summary:**

**Aspect finalize() AutoCloseable / Cleaner**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

Execution Guarantee Unpredictable Deterministic

Performance Slows GC No overhead

Java Version Deprecated (Java 9+) Recommended

Use Case Legacy cleanup Modern resource management

**( Key Takeaways:** finalize() is deprecated. Not reliable for critical resource cleanup. Prefer try-with-resources for guaranteed cleanup.**)**

Finalize Method in Java (Before Object Destruction):

---------------------------------------------------------------------

Just before destroying an object, Garbage Collector calls the object's finalize() method. The class-specific finalize() method will be executed. Example:

public class Test {

@Override

public void finalize() {

System.out.println("Finalize method called");

}

public static void main(String[] args) {

String str = "Java";

str = null;

System.gc();

System.out.println("End of main method !"); **// Output: End of main method !**

}

}

( If a String object is eligible for GC, then String class's finalize() method will be executed, not Test class's finalize() method. After finalize() execution, the object is ready to be garbage collected. If we replace a String object with a Test object, then Test class's finalize() method will be executed.)

**Explicit Call vs GC Call:** We can explicitly call the finalize() method based on our requirement. If we call finalize(), it behaves like a normal method call, and the object will not be destroyed. If Garbage Collector (GC) calls finalize(), then the object will be destroyed after method execution.

**Real-life Analogy (Servlet Life Cycle):** init(), service(), and destroy() methods are considered the life-cycle methods of a Servlet. Just before destroying the Servlet object, the Web Container calls the destroy() method for cleanup. But based on requirement, we can call destroy() manually from init() or service() methods; in that case, it behaves like a normal method call, and the Servlet object won't be destroyed.

**→** Even though an object becomes eligible for Garbage Collection (GC) multiple times, the **Garbage Collector calls the finalize() method only once for that object.** Example:

public class Test {

static Test t;

@Override

public void finalize() {

System.out.println("Finalize method called");

t = this;

}

public static void main(String[] args) throws InterruptedException {

Test t1 = new Test();

System.out.println(t1.hashCode());

t1 = null;

System.gc();

Thread.sleep(3000);

System.out.println(t.hashCode());

t = null;

System.gc();

Thread.sleep(2000);

System.out.println("End of main method !");

}

}

**Output:** 798154996

Finalize method called

798154996

End of main method !

**Garbage Collector (GC) Behavior in Java:**

**-------------------------------------------------------**

We cannot expect the exact behavior of Garbage Collector, as it varies from JVM to JVM. Hence, we can't provide exact answers for the following questions:

When exactly JVM runs GC? In which order GC identifies eligible objects? In which order GC destroys eligible objects? Whether GC destroys all eligible objects or not? What algorithm GC follows?

Whenever a program runs low on memory, JVM may request GC to run, but the **exact timing is unpredictable.** Most garbage collectors **follow a standard algorithm called "Mark and Sweep",** but it is not mandatory - different GCs may follow different algorithms.

**Mark and Sweep:** It is a garbage collection algorithm where JVM first marks all reachable (live) objects and then sweeps (deletes) all unreferenced (dead) objects from memory. Example:

public class Test {

static int count = 0;

@Override

public void finalize() {

System.out.println("Finalize method called : "+ ++count);

}

public static void main(String[] args) throws InterruptedException {

for (int i = 0; i < 1000; i++) {

Test t = new Test();

t = null;

}

}

} // In the above program, if we keep increasing the value of i in the loop, eventually a memory problem will occur. At that point, the JVM will run the garbage collector, call the finalize() method separately for each eligible object, and then destroy those objects.

**Memory Leaks in Java:** Objects that are no longer needed but are still referenced in a program are known as memory leaks.

If memory leaks exist, the program may eventually terminate by throwing an **OutOfMemoryError.** To avoid such issues, it is highly recommended to make objects eligible for garbage collection once they are no longer required.

Popular third-party memory management tools for detecting memory leaks: HP JMeter, JProbe, IBM Tivoli

**Java 8 Features**

**====================================================================================**

**Main Features of Java 8:**

**----------------------------------**

**Lambda Expressions:**  Enables functional programming by allowing you to pass functionality as a method argument.

**Functional Interfaces:** Interfaces with exactly one abstract method, enabling the use of lambda expressions (e.g., Runnable,

Comparator).

**Stream API:** Provides a powerful way to process collections (filtering, mapping, reducing) in a declarative style.

**Default Methods:** Allows interfaces to have methods with a default implementation using the default keyword.

**Method References:** A shorthand notation for calling methods directly from classes or instances.

**New Date and Time API:** A much improved, immutable, and thread-safe date-time handling API (java.time package).

**Collectors Class:** Provides methods for mutable reduction (e.g., converting Stream into List, Set, Map).

**Parallel Streams:** Easy parallel processing of collections by simply using parallelStream().

**CompletableFuture API:** Supports asynchronous programming and simplifies writing non-blocking code.

**Main Objectives of Java 8:**

**------------------------------------**

**Encourage Functional Programming:** Makes Java more flexible and expressive by adding functional features.

**Reduce Boilerplate Code:** Lambdas, streams, and method references significantly reduce code verbosity.

**Improve Performance:** Features like parallel streams and better collection processing enhance performance, especially on

multicore processors.

**Better Code Readability and Maintainability:** Declarative coding style (using Streams, Lambdas) makes code more readable

and maintainable.

**Enhanced Productivity:** Less code + powerful APIs = faster development and higher productivity.

**Backward Compatibility:** Java 8 maintained compatibility with older versions while adding powerful new features.

**What is a Lambda Expression?**

**-----------------------------------------**

A lambda expression is a short block of code that represents an instance of a functional interface (an interface with a single abstract method). It provides a clear and concise way to implement interfaces using **anonymous functions.**

**Key Features:** Enables functional programming in Java. Reduces boilerplate code. Can be stored in variables, passed as arguments, or returned from methods.

**Syntax of Lambda Expressions:** The general syntax is: **(parameters) -> { body }**

parameters: Input arguments (can be empty ()).

**-> :** Arrow operator (separates parameters from body).

**body:** Code to be executed (can be a single expression or a block). Examples:

**Scenario Lambda Expression**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

No parameters () -> System.out.println("Hello")

Single parameter (x) -> x \* x or x -> x \* x

Multiple parameters (a, b) -> a + b

Multi-line body (x, y) -> { int sum = x + y; return sum; }

**→** If the lambda body has only one statement, **{} braces and even the return keyword are optional.** Example:

(str) -> System.out.println(str);

**Type Inference:**

**----------------------**

Java Compiler can automatically infer the type of parameters in lambda expressions from the context, so we can omit explicit type declarations. Example:

(List<String> list) -> list.size(); // With type

(list) -> list.size(); // Type inferred

**Step-by-Step Conversion of Method to Lambda:**

**-----------------------------------------------------------------**

Given method: public int getLength(String str) {

return str.length();

}

Step-by-step lambda conversion:

1. Remove method name and move parameters left of ->:

(String str) -> {

return str.length();

}

1. Remove braces and return (single statement): (String str) -> str.length();
2. Apply type inference: (str) -> str.length();
3. **Final lambda: str -> str.length();**

**Functional Interface:** A Functional Interface is an interface **that contains exactly one abstract method.** **It may contain any number of default or static methods**, but only one abstract method. To indicate a functional interface, we can use the **@FunctionalInterface annotation** (optional but recommended for compile-time checking). Example:

@FunctionalInterface

interface MyFunctionalInterface {

void perform(); // Only one abstract method

}

**Using with Lambda:** MyFunctionalInterface obj = () -> System.out.println("Action performed");

obj.perform(); // Output: Action performed

**Why We Use Functional Interfaces:** To enable lambda expressions (which require a single abstract method target). To write cleaner, more concise, and readable functional-style code. Commonly used in functional programming features (e.g., Stream API, method references).

**→** If you annotate an interface with @FunctionalInterface and it does not adhere to the "exactly one abstract method" rule (either having zero or more than one abstract method), the Java compiler will issue a compile-time error. This helps developers catch mistakes early and ensures that interfaces intended for lambda expressions or method references are correctly defined.

**Functional Interface Inheritance Rule:**

**----------------------------------------------------**

If an interface extends a functional interface, and the child interface does not declare any new abstract methods, then the child interface is also a functional interface. The child interface can redeclare the same abstract method from the parent (with the same signature).

If the child interface declares any additional abstract method, it violates the rule of a functional interface, and the compiler will throw an error **if @FunctionalInterface is used**.

**Example (Valid):**

**-----------------------**

@FunctionalInterface

interface Parent {

void show();

}

@FunctionalInterface

interface Child extends Parent {

// No new abstract method; still a functional interface

void show(); // Same method as Parent

}

**Example (Invalid):** @FunctionalInterface

interface Child extends Parent {

void display(); // Compile-time error: more than one abstract method

}

**Lambda Expression to Implement Functional Interface:**

**---------------------------------------------------------------------------**

With the help of lambda expressions, we can easily implement a functional interface without creating a separate implementation class. This leads to more concise and readable code. Example:

public class Test {

public static void main(String[] args) {

Interf interf = () -> System.out.println("Lambda expression");

interf.m1();

}

}

@FunctionalInterface

interface Interf {

void m1();

}

In the above example: Interf is a functional interface with a single abstract method m1(). Instead of creating a separate class to implement Interf, we use a lambda expression: **() -> System.out.println("Lambda expression")** This is a valid implementation for m1().

**Type Inference in Lambda Expressions:**

**-----------------------------------------------------**

Type inference is the ability of the Java compiler to automatically detect the data types of lambda parameters based on the context—especially the target functional interface. Example:

interface Interf {

void add(int a, int b);

}

public class Test {

public static void main(String[] args) {

Interf interf = (num1, num2) -> System.out.println("Sum = " + (num1 + num2));

interf.add(10, 20);

}

}

**Type Inference Explained:** The lambda expression (num1, num2) -> System.out.println(...) does not explicitly declare the types of num1 and num2. The compiler looks at the target type, which is the Interf interface. From Interf, it sees that the method add(int a, int b) takes two int arguments. So, the compiler infers that num1 and num2 are both int.

Summary: Type inference allows you to omit types in lambda parameters, making the code cleaner:

// With type inference

(num1, num2) -> System.out.println("Sum = " + (num1 + num2));

// Without type inference (explicit types) (Both are valid, but using type inference reduces boilerplate.)

(int num1, int num2) -> System.out.println("Sum = " + (num1 + num2));

**Example: Implementing Runnable using Lambda to Create a Thread**

**--------------------------------------------------------------------------------------------**

This example shows how to implement the Runnable functional interface using a lambda expression, without writing a separate implementation class:

public class Test {

public static void main(String[] args) {

Runnable r1 = () -> {

for (int i = 0; i < 5; i++) {

System.out.println("Lambda Child Thread");

}

};

Thread t1 = new Thread(r1);

t1.start();

for (int i = 0; i < 5; i++) {

System.out.println("Lambda Main Thread");

}

}

}

**Explanation:** Runnable is a functional interface with a single abstract method run(). Using a lambda expression, we directly define the logic of run() without a separate class. **Thread t1 = new Thread(r1);** creates a new thread with the lambda-based Runnable. This approach makes thread creation concise and readable.

**Functional Interface – Key Concept:**

**------------------------------------------------**

A functional interface is an interface that contains only one abstract method. It is used as the target type for lambda expressions. **The purpose of introducing functional interfaces is to support lambda expressions in Java 8.** If an interface has more than one abstract method, the compiler won’t know which method the lambda expression is implementing, leading to a **"Incompatible types" compile-time error.**

**Why use @FunctionalInterface?**

**---------------------------------------------**

To enforce that an interface must have only one abstract method. If we try to add more than one abstract method, the compiler throws an error. It acts as a safeguard and improves code readability and intent. Example:

@FunctionalInterface

interface Interf {

void m1(); // Only one abstract method allowed

}

// Using lambda expression with the functional interface

Interf i = () -> System.out.println("Hello from Lambda!");

i.m1();

**Why Lambda Expression is Not a Complete Replacement for Anonymous Inner Class:**

**-------------------------------------------------------------------------------------------------------------------**

Lambda expressions simplify the syntax of implementing functional interfaces but do not cover all use cases that anonymous inner classes handle. Here's why:

**Can Implement Only Functional Interfaces:** Lambda can implement only interfaces with a single abstract method (Functional Interfaces).Anonymous Inner Classes can implement interfaces with multiple abstract methods.

**Cannot Extend Classes:** Lambda expressions cannot extend any class.Anonymous Inner Classes can extend both concrete and abstract classes.

**No Declaration of Instance Variables:** Inside a lambda expression, we cannot declare instance variables.In Anonymous Inner Classes, instance variables can be declared and used freely.

**Cannot Override Object Class Methods (like toString, equals, etc.):** Lambda expressions cannot override Object class methods directly.Anonymous Inner Classes can override any method including those from Object class.

**No Constructor Definition:** Lambda expressions don't have constructors.Anonymous Inner Classes, although nameless, can have instance initializer blocks to mimic constructor behavior.

**‘this’ Keyword Behavior:** In a lambda, this refers to the enclosing class object.In Anonymous Inner Class, this refers to the current anonymous inner class object.

**Cannot Be Instantiated:** Lambda expressions cannot be directly instantiated.Anonymous Inner Classes are instantiated immediately when declared.

**When to Use Anonymous Inner Class Over Lambda Expression: Use Anonymous Inner Class If…**

**---------------------------------------------------------------------------------------------------------------------------------**

You need to implement more than one method, extend a class (abstract or concrete), declare and manage instance variables, override Object class methods or more control and flexibility over object behavior.

**When to Prefer Lambda Expression: Use Lambda If…**

**------------------------------------------------------------------------**

You are implementing a Functional Interface. Code simplicity and readability are priorities. You don’t need instance variables or class-level behavior. You want to pass functions as arguments (Functional Programming).

**Conclusion:** Lambda expressions are lightweight, concise, and best suited for functional interfaces. But they are not a full replacement for anonymous inner classes. When you need more power and flexibility, anonymous inner classes are the better choice.

**this Keyword Behavior: Lambda vs Anonymous Inner Class:**

**---------------------------------------------------------------------------------**

In Anonymous Inner Class, this refers to the current inner class object. In Lambda Expression, this refers to the enclosing class object (i.e., outer class object). **Example:**

public class Test {

int x = 100;

public void method() {

Interf interf1 = new Interf() {

int x = 200;

@Override

public void show() {

System.out.println("Anonymous Inner Class: " + this.x);

}

};

Interf interf2 = () -> {

// No separate 'x' variable here; using outer class x

System.out.println("Lambda Expression: " + this.x);

};

interf1.show(); // Output: 200

interf2.show(); // Output: 100

}

public static void main(String[] args) {

new Test().method();

}

}

interface Interf {

void show();

}

**Explanation:** In interf1, this.x refers to the variable x = 200 of Anonymous Inner Class.In interf2, this.x refers to the variable x = 100 of the enclosing class Test because lambdas don’t define a new scope.

**Conclusion:** Anonymous Inner Classes create a new scope, so this refers to their own instance. Lambda Expressions do not create a new scope, so this refers to the enclosing class object. This is a crucial difference and often helps decide when to use which one.

**Lambda Expression: Use of Instance & Local Variables:**

**--------------------------------------------------------------------------**

Lambda expressions can access instance variables of the outer class and local variables of the enclosing method **only if they are final or effectively final.** A variable is effectively final if its value is not changed after initialization. Lambda expressions cannot modify local variables of the enclosing method. Example:

public class Test {

int instanceVar = 100;

public void method() {

int localVar = 50; // effectively final

Runnable r = () -> {

System.out.println("Instance Variable: " + instanceVar);

System.out.println("Local Variable: " + localVar); // valid

};

// localVar++; // Uncommenting this will cause compile-time error

r.run();

}

public static void main(String[] args) {

new Test().method();

}

}

**Output:** Instance Variable: 100

Local Variable: 50

**Why Local Variable Must Be Final or Effectively Final?**

**-------------------------------------------------------------------------**

**Reason:** Lambdas do not create their own scope or copy of local variables. Local variables are stored on the stack, and the lambda may use them after the method is over. To ensure thread safety and predictability, Java enforces the use of only final or effectively final local variables. This restriction avoids any unexpected behavior or side effects.

**Conclusion:** This ensures safe access without creating new memory leaks or side effects in multithreaded environments.

**Example: Lambda uses local variable after method ends:**

import java.util.function.Supplier;

public class Test {

public Supplier<String> getSupplier() {

String message = "Hello"; // effectively final

Supplier<String> supplier = () -> message;

return supplier; // returning lambda to be used later

}

public static void main(String[] args) {

Supplier<String> sup;

Test t = new Test();

sup = t.getSupplier(); // get the lambda

System.out.println(sup.get()); // Lambda still accesses 'message' even after method is over

}

}

**Output:** Hello

**Explanation:** The getSupplier() method finishes execution and returns a lambda.The local variable message is technically out of scope after the method ends.However, the lambda still remembers the value of message and can access it.This is safe only because message is effectively final — it was not modified.

**Why mutation is not allowed?**

**-----------------------------------------**

If we were allowed to modify message, we could end up with inconsistent behavior since: Lambdas do not capture the variable itself, they capture its value (a copy). The variable is stored on the stack, and once the method exits, it no longer exists. If we try to access or modify a non-final variable after its method scope ends, it will lead to undefined behavior or memory corruption.

If you try to modify the variable: String message = "Hello";

message = "Hi"; // Compile-time error if used in lambda

You’ll get: **Local variable message defined in an enclosing scope must be final or effectively final**

**Anonymous Inner Class vs Lambda Expression:**

**Anonymous Inner Class Lambda Expression**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

It is a class without a name. It is a function without a name (anonymous function).

Can extend abstract or concrete classes. Cannot extend any class.

Can implement interfaces with any Can implement only functional interfaces

number of abstract methods. (interfaces with a single abstract method).

Can declare instance variables. Cannot declare instance variables. All variables

are treated as local variables.

Can be instantiated. Cannot be instantiated.

Inside anonymous inner class, this refers Inside lambda, this refers to the enclosing

to the current anonymous inner class object. class object (not lambda itself).

Best choice when we need to override Best choice when targeting a single

multiple abstract methods. abstract method (functional interface).

Generates a separate .class file at compile time. No separate .class file is generated.

Memory is allocated on demand when object is created. Lambda expression resides in permanent memory

(method area).

**Additional Key Points:** Lambda expressions are more concise and readable compared to anonymous inner classes. Lambda improves performance, especially when used in Streams or functional-style operations. Functional interfaces like Runnable, Comparable, Callable, and custom single-method interfaces are ideal targets for lambdas.

**Advantages of Lambda Expressions in Java:**

**-----------------------------------------------------------**

**Enables Functional Programming:** Introduces a functional style to Java, allowing you to treat functions as first-class citizens.

**Concise and Readable Code:** Reduces boilerplate code significantly, making the code more concise and easier to read.

**Simplifies Anonymous Inner Class Logic:** Avoids the verbosity and complexity of anonymous inner classes for implementing

single-method interfaces.

**Treat Functions Like Values:** Allows functions/procedures to be treated as data, enabling powerful programming paradigms.

**Pass Functions as Arguments:** You can pass lambda expressions as arguments to methods, enabling higher-order functions.

**Better API Usability:** Makes it easier to work with new Java APIs and libraries, especially Streams and Collections.

**Enables Parallel Processing:** Improves performance by enabling parallel stream processing with minimal effort.

**Interface never talks about state of an object, but abstract class can:**

**---------------------------------------------------------------------------------------------**

State refers to instance variables that store data specific to an object. Interfaces cannot hold state (no instance variables).

Abstract classes can hold state using instance variables (non-static fields).

**Why? :** Interface's purpose is to define a contract (i.e., "what to do", not "how to do").Abstract classes are a halfway point between interfaces and concrete classes, so they can maintain both contract and behavior (state).Example:

interface MyInterface {

void show();

}

abstract class MyAbstractClass {

int count = 10; // this is state

abstract void show();

}

In MyAbstractClass, you can maintain count for each object (state). In MyInterface, you can’t declare such instance variables.

**Functional Interface with default methods can refer lambda expression, but abstract class can’t:**

**----------------------------------------------------------------------------------------------------------------------------------**

Lambda expressions can only be used to implement a functional interface (an interface with exactly one abstract method).

Even if the interface has multiple default or static methods, it still qualifies as long as there’s only one abstract method. Abstract classes can’t be targets for lambda expressions.

**Why abstract class can't? :** Lambda is a shortcut to implement a functional interface, not a class.Java doesn’t allow creating an object of a class (even abstract) without using new.Example:

@FunctionalInterface

interface MyFunc {

void show(); // only one abstract method

default void greet() {

System.out.println("Hello!");

}

}

public class Test {

public static void main(String[] args) {

MyFunc mf = () -> System.out.println("Showing from Lambda!");

mf.show();

mf.greet();

}

}

Here, MyFunc has a default method and still can be implemented using a lambda expression because it has only one abstract method.

**Why can't we override Object class methods in an interface?**

**----------------------------------------------------------------------------------**

Methods like equals(), hashCode(), toString() are part of java.lang.Object class. Interfaces implicitly inherit Object methods but are not allowed to override them explicitly. If Java allowed you to override them in interface, **it could cause ambiguity and conflicts** when multiple interfaces define conflicting versions of these methods. If you try:

interface MyInterface {

// Compile-time error

boolean equals(Object obj);

}

You’ll get: The interface MyInterface cannot define an abstract method equals because it would override the Object method

**(So interfaces inherit Object methods, but they can’t override them directly.)**

**Why are static methods of interfaces not available to the implemented class?**

**---------------------------------------------------------------------------------------------------------**

Static methods belong to the interface itself, not to the implementing class or the object. They are not inherited by implementing classes. So, they must be accessed using the interface name only. **Example:**

interface MyInterface {

static void greet() {

System.out.println("Hello from Interface");

}

}

class MyClass implements MyInterface {

void call() {

// greet(); Compile-time Error: Cannot find symbol

MyInterface.greet(); // Correct way to call

}

}

Why did Java design it this way?

**Avoid ambiguity:** If a class implements multiple interfaces, and all have static methods with the same name, calling them through the class would be confusing.

**Design clarity:** Static methods are **utility-like.** They should belong to the interface type, not to the instances or implementing classes.

**Predicate vs Function in Java (java.util.function Package):**

**Predicate Function**

**---------------------------------------------------------------------------------------------------------------------------------------------------------------------**

To implement conditional checks, To perform an operation and return a result,

we should go for Predicate. we should go for Function.

Predicate takes one type parameter that Function takes two type parameters: the first represents the

represents the input argument type. input type, and the second represents the return type.

**Syntax: Predicate<T> Syntax: Function<T, R>**

It defines one abstract method called test(). It defines one abstract method called apply().

Method Signature: public boolean test(T t) Method Signature: public R apply(T t)

It can only return boolean values (true or false). It can return any type of value, depending on logic.

**Extra Key Points for Notes:**

Predicate<T> is mainly used in filtering logic, for example in Stream.filter(). Function<T, R> is used when you need to transform or map one value to another, for example in Stream.map(). Both are functional interfaces, so they can be implemented using lambda expressions.

**Example for Predicate:**

Predicate<Integer> isEven = n -> n % 2 == 0;

System.out.println(isEven.test(4)); // true

**Example for Function:**

Function<String, Integer> getLength = s -> s.length();

System.out.println(getLength.apply("Java")); // 4

**Extra Insights**

**Predefined (Java Library) Classes** :

* Already compiled by Oracle/Java vendors.
* **Their .class files are packaged inside JARs** (like rt.jar in older JDKs, or modular java.base in newer JDKs).
* When you use them (String, ArrayList, etc.), the JVM loads the single compiled .class from those libraries.
* You don’t generate any new .class file for them — they already exist.

**User‑Defined Classes** : When you compile your program (javac), each top‑level class you wrote gets its own .class file.

**Example** : class A {}

class B {}

**→** Compilation produces A.class and B.class. If you remove one .class file (say B.class) but your main program references B, the JVM will throw **NoClassDefFoundError at runtime** because the bytecode is missing.

**Summary :**

* Predefined classes → single .class already exists inside JDK libraries.
* User‑defined classes → each class you write generates its own .class file.
* If a required .class file is missing at runtime, you’ll get NoClassDefFoundError.

**(In short:** library classes are shipped as one compiled .class each inside JARs; your own classes each produce their own .class file, and deleting one breaks runtime loading.**)**

**→** The **classpath** is simply the **list of folders and JAR files** where the JVM looks for **.class files** when running your program. If the JVM can’t find a class in those locations, you get a **ClassNotFoundException**.

**What is Classpath? :**

* Think of **classpath** as the **search path for Java classes**.
* It tells the **compiler** and the **JVM** where to look for :
  + Your **user‑defined classes** (.class files you compiled).
  + Any **external libraries** (JARs).
  + The **standard Java API classes** (already included in the JDK).

Without the right classpath, Java won’t know where to find the classes you reference.

**Example 1 : User‑Defined Class**

// File: Hello.java

public class Hello {

public static void main(String[] args) {

System.out.println("Hello Devi!");

}

}

**Compile :** javac Hello.java **→** Generates Hello.class.

**Run :** java Hello // Works because the current directory (.) is in the default classpath.

If you move Hello.class to another folder (say bin/), you must tell Java : **java -cp bin Hello**

Here -cp (or CLASSPATH) tells JVM: “Look inside bin/ for classes.”

**Key Difference :**

* **PATH** → tells OS where to find executables (javac, java).
* **CLASSPATH** → tells JVM where to find .class files and JARs.

**(Summary :** Classpath is the “map” the JVM uses to locate classes. If a class isn’t in the classpath, you’ll hit **ClassNotFoundException**.**)**

**→** In Java, **final variables initialized with compile‑time constants (like literals)** are treated as constants by the compiler.

* Their values are **substituted directly into the bytecode at compile time**.
* If you perform operations on two such final compile‑time constants, the **result is computed at compile time** itself.

**Example** **:**

final int A = 5;

final int B = 10;

int C = A + B; // Compiler replaces with: int C = 15;

**→** However, if a final variable is **not initialized with a compile‑time constant** (e.g., assigned from a method call or another variable), then its value is **only known at runtime**. In that case, any operation involving it will be **evaluated at runtime**.

**Example :**

final int x = getValue(); // not a compile-time constant

final int y = 10;

int z = x + y; // Computed at runtime, since x is resolved only at runtime

* **Final + compile‑time constant → replaced at compile time.**
* **Final + runtime value → resolved at runtime.**