**JAVA Basics to Advanced 2025**

**Variables:**

Variable is a place or a box to store data.

Data can be of different types

* Numbers
* Strings
* True or false
* image

We have to give a name to variable to create/define it.

```

myName = “Hello”;

myAge = 33;

avgMark = 75.8;

```

**Variable Naming Conventions:**

While you name or create a variable it's important to follow certain naming conventions to make your code readable and maintainable:

* Variable names should be descriptive and indicate the purpose of the variable.
* They must start with a letter, underscore (`\_`), or dollar sign (`$`). They cannot start with a digit.
* They can contain letters, digits, underscores, and dollar signs.
* Java is case-sensitive, so `count` and `Count` are different variables.
* By convention, variable names are usually written in `camelCase`, where the first word is in lowercase, and subsequent words start with an uppercase letter (e.g., `firstName`, `totalAmount`).
* Avoid using Java keywords (reserved words like `class`, `public`, `int`, etc.) as variable names.

**Data Types:**

We saw we can store data in variables and it can be of different types.

So, in java the data types are categorized in to two primitive and reference.

**Primitive data types:**

These data types has a fixed size and range. And stores actual values in memory.

* Byte: **byte**
  + Byte has a fixed size of 8 bits and stores whole numbers between -128 to 127.
* Short: **short**
  + Short has a fixed size of 16 bits and stores whole numbers between -32,768 to 32,767
* Integer: **int**
  + Integer has a fixed size of 32 bits and stores whole numbers between -2,147,483,648 to 2,147,483,647
* Long: **long**
  + Long has a fixed size of 64 bits and stores large whole numbers between -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
* Float: **float**
  + Float has a fixed size of 32 bits and stores single-precision floating-point numbers approximately $\pm 3.4 \times 10^{-38}$ to $\pm 3.4 \times 10^{+38}$
* Double: **double**
* Double has a fixed size of 64 bits and stores double-precision floating-point numbers approximately $\pm 1.7 \times 10^{-308}$ to $\pm 1.7 \times 10^{+308}$
* Boolean: **boolean**
  + Boolean has a fixed size of 1bit and stores `true` or `false` values.
* Character: **char**
  + Character has a fixed size of 16 bits and stores a single Unicode character range between '\u0000' to '\uffff' (0 to 65,535 in decimal)

**Examples:**

```java

byte age = 30;

short year = 2025;

int count = 100000;

long distance = 15000000000L; // 'L' suffix indicates a long literal

float pi = 3.14159f; // 'f' suffix indicates a float literal

double gravity = 9.81;

boolean isJavaFun = true;

char initial = 'J';

```

**Reference data types are:**

These data types do not store the actual value directly in memory. Instead, they store a reference (an address in memory) to the object that holds the actual data. Reference types include:

**Classes**: Blueprints for creating objects (e.g., `String`, `Scanner`, custom classes you define).

**Interfaces**: Contracts that define a set of methods that a class can implement.

**Arrays**: Collections of elements of the same data type.

**Key Points about Reference Types:**

- Their size is not fixed and depends on the size of the object they refer to.

- They are passed by reference. When you assign a reference variable to another, both variables will refer to the same object in memory.

**Examples:**

```java

String name = "Java"; // String is a class

int[] numbers = {1, 2, 3, 4, 5}; // Array of integers

Scanner scanner = new Scanner(System.in); // Scanner is a class

```

**Creation of a variable:**

There are twoparts in creating a variable, one is declaration and other is initialization.

When you are defining a variable you have to add the data type first followed by variable name then equal sign then variable itself ending with a semicolon.

```

int age = 30;

float hai = 22.3f;

double percentage = 82.55

boolean isPresent = true;

String myName = “Hello”

```

You can first declare and then initialize or do it both at the same time is allowed.

Example:

```

int rollNo; // declaration

rollNo = 24; // initialization

int rollNo = 28; both at the same time

```

Primitive data types are also called literals.

**Usage of var**

Starting from java 10 for local variables we can use var instead of specifically defining data type. Because its checked in compile time it is not going to change the status Statically typed language.

If you are using var the you have to initialize it while declaring.

Declaring first and initializing later will give you compile time error.

**Type conversion and casting**

Converting or changing the data type of a variable is not allowed but casting is.

Bigger size data type cannot be converted to smaller size but smaller size to bigger size can.

Example

```

byte b = 127;

int a = 1924;

b = a; // not allowed

a = b; // implicit conversion. This kind of assignment is ok

b = (byte) a; // explicit conversion called casting

boolean c = true;

char d = ‘G’;

int e = 7;

c = d; // not allowed

e = c; // not allowed

float f = 7.4f;

int h = (int) f; // ecplicit casting. Will lose the point value.

int a = 257;

byte b;

b = (byte) a; // do a modulus operation and convert and assign the result to it

// Result = 1

byte a = 10;

byte b = 40;

int result = a\*b; // this is called type promotion

```

**Operators**

Operators are special symbols that perform operations on one or more operands (variables or values). Java provides a rich set of operators to perform various kinds of operations. We can categorize them as follows:

1. **Arithmetic Operators**
2. **Relational operators**
3. **Logical operators**
4. **Assignment operators**
5. **Bitwise operator**
6. **Ternary operator**

**Arithmetic Operators:**

**```**

int a = 5;

int b = 8;

int result = a + b; // addition

int result = a - b; // subtraction

int result = a \* b; // multiplication

int result = a / b; // division

int result = a % b; // modulus

int result = a++; // increment (postfix)

int result = ++a; // increment (prefix)

int result = --b; // decrement (prefix)

int result = b--; // decrement (postfix)

int a = 5;

int b = a++; // b becomes 5, and then a becomes 6

System.out.println("a: " + a + ", b: " + b); // Output: a: 6, b: 5

int c = 10;

int d = ++c; // c becomes 11, and then d becomes 11

System.out.println("c: " + c + ", d: " + d); // Output: c: 11, d: 11

**```**

**Increment and Decrement Operators:**

* Postfix (x++, y--): The value of the variable is used in the current operation before it is incremented or decremented.
* Prefix (++x, --y): The value of the variable is incremented or decremented before it is used in the current operation.

**Relational Operators:**

These operators compare two operands and return a boolean value (true or false).

```Java

int x = 5;

int y = 10;

boolean result = x == y; // equal to. Result: false

boolean result = x != y; // not equal to. Result: true

boolean result = x > y; // greater than. false

boolean result = x < y; // less than. Result: true

boolean result = x <= y; // less than or equal to. Result: true

boolean result = x >= y; // greater than or equal to. Result: true

```

**Logical Operators:**

```Java

boolean x = true;

boolean y = false;

boolean result = x && y; // logical and. result: false

boolean result = !x; // logical not. result: false

boolean result = x || y; // logical or. result: false

```

**Short-circuiting in && and ||:**

* In x && y, if x is false, then y is not evaluated because the result will be false regardless of the value of y.
* In x || y, if x is true, then y is not evaluated because the result will be true regardless of the value of y.

**Assignment Operators:**

* These operators assign a value to a variable. The most common one is the simple assignment operator (=).
* Java also provides shorthand assignment operators that combine an arithmetic operation with assignment.

```Java

//Simple assignment

int x = 10; // equivalent to x = 10

//Add and Assign

x += 5; // equivalent to x = x + 5

//subtract and assign

x -= 5; // equivalent to x = x – 5

//Multiply and assign

x \*= 5; // equivalent to x = x \* 5

//Divide and assign

x /= 5; // equivalent to x = x / 5

//Modulus and assign

x %= 5; // equivalent to x = x % 5

```

**Bitwise Operators:**

These operators perform operations on the individual bits of integer operands (byte, short, int, long).

Example (if a=5 (0101), b=3 (0011))

```Java

a & b // Bitwise AND. result binary 0001. Result (decimal) 1

` // Bitwise OR. result binary `a \. Result (decimal) b`

a ^ b // Bitwise XOR. result binary 0110. Result (decimal) 6

~a // Bitwise NOT. result binary 11111...1010 (depends on bit length). Result (decimal) -6

a << 2 // Left shift. result binary 010100...1010 Result (decimal) 20

a >> 1 // Right shift. result binary 0010...1010 Result (decimal) 2

a >>> 1 // Unsigned right shift. result binary 0010...1010 Result (decimal) 2

```

**Understanding Bitwise Shifts:**

* **Left Shift (<<):** Shifts the bits to the left by the specified number of positions. Each left shift effectively multiplies the number by 2.
* **Right Shift (>>):** Shifts the bits to the right by the specified number of positions. The sign bit (most significant bit) is preserved. Each right shift effectively divides the number by 2 (integer division).
* **Unsigned Right Shift (>>>):** Shifts the bits to the right by the specified number of positions, and the leftmost bits are filled with zeros, regardless of the original sign bit.

**Ternary Operator (Conditional Operator):**

This is a shorthand for a simple if-else statement. Its syntax is:

```Java

// condition ? value\_if\_true : value\_if\_false;

int num1 = 10; int num2 = 5; int max = (num1 > num2) ? num1 : num2; // If num1 > num2, max = num1, else max = num2 System.out.println("The maximum is: " + max); // Output: The maximum is: 10

```

**Operator Precedence:**

Operators have a defined precedence that determines the order in which they are evaluated in an expression. It's often a good practice to use parentheses to explicitly control the order of operations and make your code clearer. Here's a general order of precedence (from highest to lowest):

1. Postfix increment/decrement (x++, x--)
2. Prefix increment/decrement (++x, --x), Unary plus/minus (+x, -x), Logical NOT (!), Bitwise NOT (~)
3. Multiplication (\*), Division (/), Modulus (%)
4. Addition (+), Subtraction (-)
5. Shift operators (<<, >>, >>>)
6. Relational operators (<, >, <=, >=), instanceof
7. Equality operators (==, !=)
8. Bitwise AND (&)
9. Bitwise XOR (^)
10. Bitwise OR (|)
11. Logical AND (&&)
12. Logical OR (||)
13. Ternary operator (?:)
14. Assignment operators (=, +=, -=, \*=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=)

Understanding operators and their precedence is crucial for writing correct and predictable Java code.

**Control Flow**

Java provides several control flow statements that allow you to control the sequence of execution in your program. These can be broadly categorized into:

**Conditional Statements:** These statements allow you to execute different blocks of code based on whether a certain condition is true or false.

* **if statement:** Executes a block of code if a specified condition is true.

```Java

int age = 20;

if (age >= 18) { System.out.println("You are eligible to vote."); }

```

* **if-else statement:** Executes one block of code if a condition is true and another block if the condition is false.

```Java

int temperature = 15;

if (temperature > 20) {

System.out.println("It's warm.");

} else {

System.out.println("It's cool.");

}

```

* **if-else if-else statement:** Allows you to check multiple conditions in sequence. The first condition that evaluates to true will have its corresponding block of code executed. The final else block (if present) is executed if none of the preceding conditions are true.

```Java

int score = 75;

if (score >= 90) { System.out.println("Grade: A"); } else if (score >= 80) { System.out.println("Grade: B"); } else if (score >= 70) { System.out.println("Grade: C"); } else if (score >= 60) { System.out.println("Grade: D"); } else { System.out.println("Grade: F"); }

```

* **switch statement:** Provides a way to execute different blocks of code based on the value of a single variable (or expression). The variable must be of a primitive type (byte, short, int, char), an enum, or a String (since Java 7).

```Java

int dayOfWeek = 3;

String dayName;

switch (dayOfWeek) {

case 1:

dayName = "Monday";

break; // Important to exit the switch block

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day";

break;

}

System.out.println("Day " + dayOfWeek + " is " + dayName);

```

**Key points about switch statements:**

* + The case labels must be constant expressions of the same type as the switch variable.
  + The break statement is crucial to exit the switch block after a matching case is found. If break is omitted, execution will "fall through" to the next case. This can sometimes be intentional but is often a source of bugs.
  + The default case is optional and is executed if none of the case values match the switch variable's value. It's good practice to include a default case to handle unexpected values.

**Looping Statements:** These statements allow you to execute a block of code repeatedly as long as a certain condition is true.

**for loop:** Used when you know in advance how many times you want to execute a block of code. It consists of three parts:

* **Initialization:** Executed only once at the beginning of the loop. Typically used to initialize a loop counter.
* **Condition:** Evaluated before each iteration. The loop continues as long as the condition is true.
* **Increment/Decrement (Update):** Executed after each iteration. Typically used to update the loop counter.

```Java

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

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

}

```

You can also have more complex for loops with multiple initializations or updates, but it's generally better to keep them simple for readability.

**Enhanced for loop (for-each loop):** This is a simplified way to iterate over collections (like arrays and lists).

```Java

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

for (int number : numbers) {

System.out.println("Number: " + number);

}

```

**while loop:** Executes a block of code as long as a specified condition is true. The condition is checked *before* each iteration.

```Java

int count = 0;

while (count < 3) {

System.out.println("Count is: " + count);

count++;

}

```

**do-while loop:** Similar to the while loop, but the condition is checked *after* each iteration. This guarantees that the loop body will execute at least once, even if the condition is initially false.

```Java

int i = 0;

do {

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

i++;

} while (i < 2);

```

**Jump Statements:** These statements allow you to alter the normal flow of execution within loops or switch statements.

**break statement:**

* + When used inside a loop (for, while, do-while), it immediately terminates the loop and transfers control to the statement immediately following the loop.
  + When used inside a switch statement, it terminates the switch block.

```Java

for (int i = 0; i < 10; i++) {

if (i == 5) {

break; // Exit the loop when i is 5

}

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

}

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

```

**continue statement:** When used inside a loop, it skips the rest of the current iteration and proceeds to the next iteration of the loop.

```Java

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

if (i == 2) {

continue; // Skip the rest of this iteration when i is 2

}

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

}

```

Understanding and using these control flow statements effectively is essential for creating programs that can make decisions and perform repetitive tasks.

**Arrays**

An **array** in Java is a fixed-size, ordered collection of elements of the same data type. Arrays provide a way to store and access multiple values using a single variable name and an index (position).

**Key Characteristics of Arrays:**

* **Fixed Size:** Once an array is created, its size cannot be changed.
* **Homogeneous:** All elements in an array must be of the same data type (primitive or reference).
* **Ordered:** Elements in an array are stored in a specific order, and each element has an index starting from 0.
* **Contiguous Memory Allocation:** Typically, array elements are stored in contiguous memory locations, which allows for efficient access to elements.

**Declaration and Initialization of Arrays:**

There are several ways to declare and initialize arrays in Java:

**1. Declaration followed by Initialization:**

```Java

// Declare an array of integers named 'numbers' that can hold 5 elements

int[] numbers;

// Allocate memory for 5 integers

numbers = new int[5];

// Initialize the elements of the array using their index (0-based)

numbers[0] = 10;

numbers[1] = 20;

numbers[2] = 30;

numbers[3] = 40;

numbers[4] = 50;

```

You can also combine the declaration and allocation in one step:

Java

int[] scores = new int[10]; // Declare and allocate memory for 10 integers

**2. Declaration and Initialization with Initial Values:**

You can directly initialize an array with values during declaration using curly braces {}:

```Java

int[] ages = {25, 30, 22, 28, 27}; // An array of 5 integers

String[] names = {"Alice", "Bob", "Charlie"}; // An array of 3 Strings

```

In this case, the size of the array is automatically determined by the number of elements provided within the curly braces.

**Accessing Array Elements:**

You can access individual elements of an array using their index, which starts from 0 for the first element and goes up to array.length - 1 for the last element.

```Java

int[] values = {100, 200, 300};

System.out.println("First element: " + values[0]); // Output: First element: 100

System.out.println("Second element: " + values[1]); // Output: Second element: 200

System.out.println("Third element: " + values[2]); // Output: Third element: 300

```

**Changing the array value:**

```Java

class Hello {

    public static void main(String[] args) {

        int nums[] = {1,2,3,4,5};

        nums[2] = 10;

        System.out.println(nums[2]);

    }

}

```

**Array Length:**

You can get the number of elements in an array using the length property (note that it's a property, not a method, so you don't use parentheses).

```Java

int[] data = {1, 2, 3, 4, 5, 6};

System.out.println("Length of the array: " + data.length); // Output: Length of the array: 6

```

**Iterating Through Arrays:**

You can use loops to process all the elements of an array. The for loop and the enhanced for loop are commonly used for this purpose.

**Using a traditional for loop:**

```Java

int[] numbers = {10, 20, 30, 40, 50};

for (int i = 0; i < numbers.length; i++) {

System.out.println("Element at index " + i + ": " + numbers[i]);

}

```

**Using an enhanced for loop (for-each loop):**

This provides a more concise way to iterate through the elements of an array without explicitly managing indices.

```Java

int[] scores = {85, 92, 78, 95, 88};

for (int score : scores) {

System.out.println("Score: " + score);

}

```

**Multi-dimensional Arrays:**

Java also supports multi-dimensional arrays, which are arrays of arrays. The most common type is a two-dimensional array, which can be thought of as a table with rows and columns.

**Declaration and Initialization of a 2D Array:**

```Java

// Declare a 2D array with 3 rows and 4 columns

int[][] matrix = new int[3][4];

// Initialize elements

matrix[0][0] = 1;

matrix[0][1] = 2;

matrix[1][2] = 7;

// ... and so on

// Declare and initialize with values

int[][] grid = {

{1, 2, 3}, // Row 0

{4, 5, 6}, // Row 1

{7, 8, 9} // Row 2

};

```

**Accessing Elements in a 2D Array:**

You need two indices to access an element in a 2D array: the row index and the column index.

```Java

System.out.println("Element at row 1, column 2: " + grid[1][2]); // Output: 6

```

**Iterating Through a 2D Array:**

You typically use nested loops to iterate through the rows and columns of a 2D array.

```Java

int[][] table = {

{10, 20},

{30, 40},

{50, 60}

};

for (int i = 0; i < table.length; i++) { // Iterate through rows

for (int j = 0; j < table[i].length; j++) { // Iterate through columns in the current row

System.out.print(table[i][j] + " ");

}

System.out.println(); // Move to the next line after each row

}

```

Java also allows for arrays with more than two dimensions, but they are less commonly used in basic programming.

**Jagged array:**

```Java

Int arr [ ] [ ] = new int [4] [ ]

```

These are arrays where you don’t specify the no of elements in each row.

**Drawback of array**: Because array is an object and get stored in heap memory once the size is defined then it cannot be changed.

Not good if you have multiple types of elements.

If you know the size of array and is fixed then go for arrays else go for collections which we will see later.

**Out of bound exception**

```Java

Int arr [ ] = new int [6];

arr [ 2 ];

arr [ 4 ];

arr [ 1 ];

arr [ 5 ];

for(int i=0; i<7; i++)

{

System.out.println(arr(i));

}

// above will cause a exception error on runtime.

// To solve it use the loop like below.

for(int i=0; i<arr.length; i++)

{

System.out.println(arr(i));

}

```

**Object Array**

```Java

class Student

{

int roolNo;

String name;

int marks;

}

public class Demo

{

public static void main(String[] args)

{

Student s1 = new Student();

s1.rollNo = 1;

s1.name = “Hello”;

s1.marks = 88;

Student s2 = new Student();

s2.rollNo = 2;

s2.name = “Mello”;

s2.marks = 89;

Student s3 = new Student();

s3.rollNo = 3;

s3.name = “Yello”;

s3.marks = 78;

Student students [ ] = new Student [ 3 ];

student [0] = s1;

student [1] = s2;

student [2] = s3;

for(int i=0; i<students.length; i++)

{

System.out.println(students [ i ].name + “: “ + students [ i ].marks);

}

}

}

**For each loop for arrays**

```Java

Int arr [ ] = new int [6];

arr [ 2 ];

arr [ 4 ];

arr [ 1 ];

arr [ 5 ];

for (int n: arr)

{

System.out.println(n);

}

```

**Using the same enhanced for loop in Student object array**

```Java

Student students [ ] = new Student [ 3 ];

student [0] = s1;

student [1] = s2;

student [2] = s3;

for(Student stud: students)

{

System.out.println(stud.name + “: “ + stud.marks);

}

```

Arrays are a fundamental data structure in Java and are used extensively for storing and manipulating collections of data. However, it's important to remember their fixed size limitation. For more dynamic collections, Java provides the **Collections Framework**, which we will explore later.

**JDK, JVM, JRE**

JDK is the tool collection that you need to have to run your java code.

JVM is the virtual machine where you run the code.

JVM needs an environment to run the code and is called JRE – Java Runtime Environment.

When we install JDK we get updated versions of JRE and JVM.

Normally client systems have JRE and JVM.

JVM

JRE

JDK

**Object Oriented Programing Concepts.**

Object-Oriented Programming (OOP) Principles in Java. OOP is a programming paradigm centered around the concept of "objects," which are instances of "classes." Java is fundamentally an object-oriented language, and understanding these principles is essential for writing well-structured, maintainable, and scalable code.

Every object has property and behaviour.

To create an object we need a class, the blue print.

JVM creates object in java.

Class code get converted to byte code and it goes to JVM to create an object.

**Class and object creation**

```Java

class Calculator

{

// variables inside a class at the are called Instance variable

int ref = 4;

// Methods

public void inAdd()

{

System.out.println(“Inside add”);

}

public int add(int n1, int n2)

{

int result = n1 + n2;

Return result;

}

}

public class Demo

{

public static void main(String[] args)

{

// Stage 1

int num1 = 3;

int num2 = 4;

int result = num1 + num2; // action or behaviour

System.out.println(result);

// Stage 2

// To use the above created class we need to create an object in our

current class.

Calculator calc = new Calculator();

// Calculator is the class name, calc is the reference variable.

// Created an object of type Calculator using Calculator class

(Blueprint).

// now calling the method inside Calculator class using newly created object.

calc.inAdd();

calc add( num1, num2 );

}

}

```

**If else with class and method:**

```Java

class Shop

{

// Methods

public void enterTheStore()

{

System.out.println(“Welcome to My Book Shop”);

}

public String buyMeABook(int cost)

{

if(cost >= 15)

return “Thank you for the purchase. This is your New Book”;

Else

return “Sorry You have to pay more than 15.rs”;

}

}

public class Demo

{

public static void main(String[] args)

{

Shop obj = new Shop();

Obj.enterTheStore();

String buyBook = buyMeABook(15);

System.out.println(buyBook);

}

}

```

**Methods Overloading:**

You can have methods with same name but need have difference in no of parameters or type. This is called method overloading.

Return type dosent mater but the parameters and its type.

Every method will have it own stack.

```Java

class Calculator

{

// Instance variable

int num = 22;

// Methods

public int add(int num1, int num2)

{

return n1 + n2; // n1 and n2 are called local variables

}

public int add(int num1, int num2, int num3)

{

return n1 + n2 + n3;

}

public double add(double num1, double num2)

{

return n1 + n2;

}

}

public class Demo

{

public static void main(String[] args)

{

int n1 = 11; // local variables

int n2 = 12;

double n3 = 13.5;

double n4 = 14.4;

Calculator obj = new Calculator();

Calculator obj2 = new Calculator();

int twoParams = obj.add(n1, n2);

int threeParams = obj.add(n1, n2);

double twoParamsDiffType = obj.add(n3, n4);

System.out.println(twoParams);

System.out.println(threeParams);

System.out.println(twoParamsDiffType);

// manipulating the reference variable inside Calculator class

// You can change the value of instance variables like below.

obj.num = 8;

System.out.println(obj.num);

System.out.println(obj1.num);

}

}

```

**What is happening?**

Here the main function has a stack memory where the instance variable like n1, n2, n3, n4 and twoParams, threeParms, twoParamsDiffType get stored.

But when you create an object the reference variable gets stored in the main stack as key and the reference address is stored as value.

This reference address points to the object created in the heap memory.

Where as the instance variable in the method get stored in the top of the object created in heap memory.

And below it will have other methods in that object.

**Strings**

In Java, a **String** is a sequence of characters. However, unlike some other programming languages where strings are just arrays of characters, in Java, String is a **class** (specifically, java.lang.String). This means that strings in Java are objects, and the String class provides various methods for manipulating and working with text.

**Key Characteristics of Java Strings:**

* **Immutable:** Once a String object is created, its value cannot be changed. Any operation that appears to modify a string actually creates a new String object.
* **Object:** Strings are instances of the String class.
* **Unicode:** Java strings support Unicode characters, allowing them to represent text in various languages.

There are several ways to create String objects in Java:

**String Literals:**

The most common way is to use a string literal, which is a sequence of characters enclosed in double quotes (").

```Java

String message = "Hello, Java!";

String empty = "";

String singleChar = "A";

```

When you use a string literal, the Java Virtual Machine (JVM) often optimizes memory usage by using a "string pool." If a string literal with the same content already exists in the pool, the new variable will refer to the existing string object instead of creating a new one.

**Using the new keyword:**

You can explicitly create a String object using the new keyword and one of the String class constructors:

Because String is a class.

You can use String with + operator.

```Java

String greeting = new String("Welcome");

String anotherGreeting = new String(new char[ ] {'W', 'e', 'l', 'c', 'o', 'm', 'e'});

String name = new String(); // empty object in heap

String name = new String(“John”); // now has a value

System.out.println(“Hello: ” + name);

// there are certain methods associated with Strings which we can use.

System.out.println (name.charAt(1));

// prints the character at index 1

System.out.println (name.concat(“Honai”));

// append Honai to the name

```

While this approach works, it's generally less efficient than using string literals because it always creates a new String object in the heap, even if an identical string exists in the string pool.

**Basic String Operations:**

The String class provides a rich set of methods for performing various operations on strings. Here are some of the most commonly used ones:

* length(): Returns the number of characters in the string.

```Java

String text = "Java";

int length = text.length(); // length will be 4

System.out.println("Length: " + length);

String word = "Example";

char firstChar = word.charAt(0);

// firstChar will be 'E'

char thirdChar = word.charAt(2);

// thirdChar will be 'a'

System.out.println("First char: " + firstChar + ", Third char: " + thirdChar);

String firstName = "John"; String lastName = "Doe"; String fullName = firstName.concat(" ").concat(lastName); // fullName will be "John Doe" System.out.println("Full name: " + fullName);

String fullName2 = firstName + " " + lastName; // Equivalent to the above System.out.println("Full name 2: " + fullName2);

```

**substring(int beginIndex):** Returns a new string that is a substring of the original string, starting from the specified beginIndex (inclusive) to the end of the string.

```Java

String message = "Hello, World!";

String sub1 = message.substring(7); // sub1 will be "World!"

System.out.println("Substring 1: " + sub1);

```

**substring(int beginIndex, int endIndex):** Returns a new string that is a substring of the original string, starting from the beginIndex (inclusive) and ending at the endIndex (exclusive).

```Java

String message2 = "Programming"; String sub2 = message2.substring(0, 4); // sub2 will be "Prog" System.out.println("Substring 2: " + sub2);

```

**equals(Object anObject):** Compares the current string to the specified object. It returns true if the strings have the same sequence of characters, and false otherwise. **Important:** Use equals() to compare string content, not ==, which compares object references.

``Java

String str1 = "Java";

String str2 = "Java";

String str3 = new String("Java");

System.out.println(str1.equals(str2)); // Output: true System.out.println(str1.equals(str3)); // Output: true

System.out.println(str1 == str2); // Output: true (due to string pool) System.out.println(str1 == str3); // Output: false (different objects in memory)

```

**equalsIgnoreCase(String anotherString):**

Similar to equals(), but it ignores case differences during comparison.

```Java

String s1 = "java";

String s2 = "JAVA";

System.out.println(s1.equalsIgnoreCase(s2)); // Output: true

```

**toLowerCase():**

Returns a new string with all characters converted to lowercase.

``Java

String upperCase = "HELLO";

String lowerCase = upperCase.toLowerCase(); // lowerCase will be "hello"

System.out.println("Lowercase: " + lowerCase);

```

**toUpperCase():**

Returns a new string with all characters converted to uppercase.

```Java

String lowerCase2 = "world";

String upperCase2 = lowerCase2.toUpperCase(); // upperCase2 will be "WORLD"

System.out.println("Uppercase: " + upperCase2);

```

**trim():**

Returns a new string with leading and trailing whitespace removed.

``Java

String spaced = " Extra spaces ";

String trimmed = spaced.trim(); // trimmed will be "Extra spaces"

System.out.println("Trimmed: \"" + trimmed + "\""); // Use quotes to show the absence of spaces

```

**replace(char oldChar, char newChar):**

Returns a new string where all occurrences of oldChar are replaced with newChar.

```Java

String text = "banana";

String replaced = text.replace('a', 'o'); // replaced will be "bonono"

System.out.println("Replaced: " + replaced);

```

**indexOf(String str):**

Returns the index of the first occurrence of the specified substring within the string, or -1 if the substring is not found.

```Java

String sentence = "This is a sentence.";

int index = sentence.indexOf("is"); // index will be 2

System.out.println("Index of 'is': " + index);

```

**lastIndexOf(String str):**

Returns the index of the last occurrence of the specified substring within the string, or -1 if the substring is not found.

```Java

String sentence2 = "This is also is a sentence.";

int lastIndex = sentence2.lastIndexOf("is"); // lastIndex will be 11

System.out.println("Last index of 'is': " + lastIndex);

```

**startsWith(String prefix):**

Returns true if the string starts with the specified prefix.

```Java

String greeting = "Hello, World!";

boolean startsWithHello = greeting.startsWith("Hello"); // true

System.out.println("Starts with 'Hello': " + startsWithHello);

```

**endsWith(String suffix):**

Returns true if the string ends with the specified suffix.

```Java

String filename = "document.txt";

boolean endsWithTxt = filename.endsWith(".txt"); // true

System.out.println("Ends with '.txt': " + endsWithTxt);

```

**StringBuilder and StringBuffer for Mutable Strings:**

Because String objects are immutable, performing many string manipulations using concatenation or replacement can be inefficient as it creates new String objects repeatedly. For scenarios where you need to perform frequent modifications to strings, Java provides the StringBuilder and StringBuffer classes in the java.lang package.

* **StringBuilder:** A mutable sequence of characters. It is not thread-safe, which makes it faster than StringBuffer in single-threaded environments.
* **StringBuffer:** A mutable sequence of characters. It is thread-safe (methods are synchronized), making it safe to use in multithreaded environments but potentially slower than StringBuilder.

You can create and modify StringBuilder or StringBuffer objects using methods like append(), insert(), delete(), replace(), and reverse(). You can then convert them back to a String using the toString() method.

```Java

StringBuilder sb = new StringBuilder("Java");

sb.append(" Programming"); // sb now contains "Java Programming"

sb.insert(5, " Language"); // sb now contains "Java Language Programming"

String finalString = sb.toString(); // Convert back to String

System.out.println("Final string: " + finalString);

StringBuffer sb = new StringBuffer("Java");

System.out.println(sb.capacity()); // returns the allocated size for that string

sb.append(“ Cool”);

System.out.println(sb) // prints “Java Cool”

sb.deleteCharAt ( 2 ); // deletes the character at index 2

sb.insert(0, “This ”); // insert This before Java.

```

**String constant pool**

If you have string with same value the only one will be stored in the string constant pool stack in heap.

Variables in the main stack will have the same reference id.

Understanding the String class and its methods is crucial for working with text data in Java. Remember the immutability of String and when to use StringBuilder or StringBuffer for mutable string operations.

This concludes our exploration of basic data types in Java. We've covered primitive types, variables, operators, control flow, arrays, and strings.

**Static Keyword**

Alright class, settle in! Today, we're going to unravel one of Java's fundamental keywords: **static**. This keyword often causes a bit of confusion, but once you grasp its core concept, it becomes incredibly powerful and clear.

Think of static as defining something that **belongs to the class itself**, rather than to any specific object (or instance) of that class. It's about "class-level" vs. "object-level" members.

Let's dive deep into the **why, when, and how** of using the static keyword in Java.

**The Core Concept: Class vs. Object**

Before static, let's quickly review the basics:

* **Class:** A blueprint or a template for creating objects. It defines what objects of that type will have (attributes/fields) and what they can do (behaviors/methods).
* **Object (Instance):** A concrete realization of a class. When you create an object (e.g., MyClass myObject = new MyClass();), you're making a unique entity based on that blueprint. Each object has its own set of instance variables.

Now, imagine we want something that all objects of a class **share**, or something that exists even if no objects of that class have been created yet. That's where static comes in!

**1. static Fields (Class Variables)**

When you declare a field (a variable) with the static keyword, it becomes a **class variable**.

* **WHY?**
  + **Shared Data:** You need a piece of data that is shared by *all* instances of a class. If one object changes it, all other objects immediately see that change.
  + **Global Access (within the class's scope):** You need data that exists and is accessible even when no objects of the class have been created.
  + **Constants:** For values that never change and are associated with the class itself (often combined with final).
* **HOW?**
  + You declare the field with the static modifier before its type.
  + Syntax: static DataType fieldName;
* **WHEN?**
  + **Counting Instances:** To keep track of how many objects of a class have been created.
  + **Constants:** Mathematical constants (Math.PI), application-wide configuration values (DatabaseConfig.URL).
  + **Shared Resources:** A common pool of resources for all objects.

**Example: Counting Instances & Constants**

Java

class Car {

// static field: belongs to the Car class, shared by all Car objects

private static int numberOfCarsCreated = 0;

// static final field: a constant belonging to the Car class

public static final String DEFAULT\_COLOR = "Red";

String model; // instance field: belongs to each Car object

public Car(String model) {

this.model = model;

numberOfCarsCreated++; // Increment the shared counter each time a car is created

System.out.println("Car '" + model + "' created. Total cars: " + numberOfCarsCreated);

}

public static int getNumberOfCars() {

return numberOfCarsCreated;

}

public void displayInfo() {

System.out.println("Model: " + this.model + ", Default Color: " + Car.DEFAULT\_COLOR);

}

}

public class StaticFieldDemo {

public static void main(String[] args) {

// Accessing static fields directly via the class name

System.out.println("Initial cars created: " + Car.numberOfCarsCreated); // Output: Initial cars created: 0

System.out.println("Default Car Color: " + Car.DEFAULT\_COLOR); // Output: Default Car Color: Red

Car car1 = new Car("Sedan"); // Creates 1st car

Car car2 = new Car("SUV"); // Creates 2nd car

// Both objects share the same static variable

System.out.println("Car 1's view of cars created: " + car1.numberOfCarsCreated); // Discouraged practice!

System.out.println("Car 2's view of cars created: " + car2.numberOfCarsCreated); // Discouraged practice!

// Best practice: Access static fields via the class name

System.out.println("Total cars (from class): " + Car.numberOfCarsCreated); // Output: Total cars (from class): 2

car1.displayInfo(); // Output: Model: Sedan, Default Color: Red

}

}

**Key Points about static fields:**

* There is only **one copy** of a static field per class, regardless of how many objects are created.
* They are loaded into memory when the class is loaded (usually when the program starts or the class is first referenced).
* **Best Practice:** Access static fields using the **class name** (e.g., Car.numberOfCarsCreated). Accessing them via an object reference (car1.numberOfCarsCreated) works but is misleading and discouraged because it suggests the field belongs to the object, which it doesn't.

**2. static Methods (Class Methods)**

When you declare a method with the static keyword, it becomes a **class method**.

* **WHY?**
  + **Utility Functions:** You need a method that performs an action but doesn't require any specific state of an object. It operates purely on its inputs or on static fields.
  + **Object Creation:** To create instances of the class (often called "factory methods").
  + **Entry Point:** The main method must be static because the JVM needs to call it to start the program without first creating an object.
* **HOW?**
  + You declare the method with the static modifier before its return type.
  + Syntax: static ReturnType methodName(Parameters) { ... }
* **WHEN?**
  + **Helper/Utility Classes:** Classes like java.lang.Math (Math.sqrt(), Math.max()), java.util.Arrays (Arrays.sort()).
  + **Factory Methods:** Methods that return an instance of the class (e.g., Integer.valueOf(10)).
  + **main method:** The entry point of your Java application.
  + **Operating on Static Fields:** Methods that primarily manipulate or return the value of static fields.

**Example: Utility Method & Main Method**

Java

class Calculator {

// Static method: does not depend on any specific Calculator object's state

public static int add(int a, int b) {

return a + b;

}

// Static method: can access static fields

public static int multiplyBy(int number) {

// You could imagine a static constant here

// public static final int FACTOR = 10;

// return number \* FACTOR;

return number \* 2;

}

// Instance method: depends on the object's state (e.g., a 'result' field)

private int result; // instance field

public void setResult(int r) {

this.result = r;

}

public int getResult() {

return this.result;

}

}

public class StaticMethodDemo {

// The main method MUST be static because the JVM calls it without an object.

public static void main(String[] args) {

// Accessing static methods directly via the class name

int sum = Calculator.add(5, 7);

System.out.println("5 + 7 = " + sum); // Output: 5 + 7 = 12

int product = Calculator.multiplyBy(8);

System.out.println("8 \* 2 = " + product); // Output: 8 \* 2 = 16

// You CANNOT call non-static methods or access non-static fields directly from a static method

// (without an object reference)

// Calculator.setResult(10); // Compile error: Non-static method 'setResult(int)' cannot be referenced from a static context

// To use an instance method, you need an object:

Calculator myCalc = new Calculator();

myCalc.setResult(100);

System.out.println("MyCalc result: " + myCalc.getResult()); // Output: MyCalc result: 100

}

}

**Key Points about static methods:**

* They can only directly access static fields and call other static methods of the same class.
* They **cannot** directly access non-static (instance) fields or call non-static (instance) methods because they don't operate on a specific object and thus have no this reference.
* **Best Practice:** Access static methods using the **class name** (e.g., Calculator.add()).

**3. static Initialization Blocks**

A static block (also known as a static initializer) is a special block of code that is executed only once when the class is loaded into the JVM.

* **WHY?**
  + **Complex Static Initialization:** To initialize static fields that require more complex logic than a simple one-line assignment.
  + **Resource Loading:** To load resources (like native libraries, configuration files) that are needed by the class.
  + **One-time Setup:** To perform any setup that needs to happen only once when the class is first used.
* **HOW?**
  + Simply use the static keyword followed by a code block: static { ... }
* **WHEN?**
  + **Loading Drivers:** For example, loading a JDBC driver for a database connection.
  + **Populating Static Data Structures:** Creating and populating a static Map or List with initial data.
  + **Complex static final initialization:** When a static final field needs more than a simple literal value.

**Example:**

Java

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

class AppConfig {

public static final String APP\_VERSION;

public static final String STARTUP\_TIME;

public static String DATABASE\_URL = "jdbc:mysql://localhost:3306/mydb"; // Simple static field

// Static initialization block

static {

System.out.println("--- AppConfig: Static block executed ---");

APP\_VERSION = "1.0.0-BETA"; // Initialize a static final field

STARTUP\_TIME = LocalDateTime.now().format(DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss"));

// More complex logic, e.g., loading a driver

try {

Class.forName("com.mysql.cj.jdbc.Driver"); // Example of loading a driver

System.out.println("MySQL JDBC Driver loaded.");

} catch (ClassNotFoundException e) {

System.err.println("Error loading JDBC Driver: " + e.getMessage());

}

}

public AppConfig() {

System.out.println("AppConfig: Constructor executed.");

}

}

public class StaticBlockDemo {

public static void main(String[] args) {

System.out.println("Main method started.");

// Accessing a static member causes the class to be loaded, triggering the static block

System.out.println("App Version: " + AppConfig.APP\_VERSION);

System.out.println("App Startup Time: " + AppConfig.STARTUP\_TIME);

System.out.println("Database URL: " + AppConfig.DATABASE\_URL);

// Creating an object also causes the class to be loaded if not already

AppConfig config = new AppConfig();

System.out.println("AppConfig object created.");

System.out.println("Main method finished.");

}

}

**Output:**

Main method started.

--- AppConfig: Static block executed ---

MySQL JDBC Driver loaded.

App Version: 1.0.0-BETA

App Startup Time: 2025-05-30 11:07:31 // (or current time)

Database URL: jdbc:mysql://localhost:3306/mydb

AppConfig: Constructor executed.

AppConfig object created.

Main method finished.

**Key Points about static blocks:**

* They execute **only once** when the class is first loaded into memory.
* They run *before* any static methods are called (including main) and *before* any objects of the class are created.
* They are executed in the order they appear in the class if there are multiple static blocks.

**4. static Nested Classes (Static Inner Classes)**

A static nested class is a nested class declared with the static modifier.

* **WHY?**
  + **Logical Grouping:** To logically group a helper class with its outer class without implying a direct instance-level relationship.
  + **Encapsulation:** To keep the helper class's implementation private to its outer class, if desired.
  + **Independence:** The nested class does not need access to the non-static (instance) members of the outer class.
* **HOW?**
  + You declare the nested class with the static modifier.
  + Syntax: class OuterClass { static class NestedStaticClass { ... } }
* **WHEN?**
  + **Helper Data Structures:** When a data structure is conceptually part of another class but doesn't need to depend on an *instance* of that class (e.g., a Node in a LinkedList implementation, which can exist without a specific LinkedList object).
  + **Builder Pattern:** Often used to create complex objects, where the builder class is static nested.
  + **Grouping Utility Classes:** If a utility class is strongly related to another class but doesn't need outer instance context.

**Example: A Simple Builder Pattern**

Java

class Pizza {

private String size;

private boolean pepperoni;

private boolean mushrooms;

// Private constructor, so Pizza objects can only be created via the Builder

private Pizza(Builder builder) {

this.size = builder.size;

this.pepperoni = builder.pepperoni;

this.mushrooms = builder.mushrooms;

}

public void display() {

System.out.println("Pizza: Size=" + size +

", Pepperoni=" + pepperoni +

", Mushrooms=" + mushrooms);

}

// Static nested class: It's a helper class for Pizza

// It doesn't need a specific 'Pizza' object to exist to create a 'Builder' object.

public static class Builder {

private String size;

private boolean pepperoni = false; // Default values

private boolean mushrooms = false;

public Builder(String size) {

this.size = size;

}

public Builder withPepperoni() {

this.pepperoni = true;

return this; // Return builder for chaining

}

public Builder withMushrooms() {

this.mushrooms = true;

return this;

}

public Pizza build() {

return new Pizza(this); // Creates the Pizza object

}

}

}

public class StaticNestedClassDemo {

public static void main(String[] args) {

// Instantiate the static nested class directly using OuterClass.NestedClass

Pizza smallVeggie = new Pizza.Builder("Small")

.withMushrooms()

.build();

smallVeggie.display(); // Output: Pizza: Size=Small, Pepperoni=false, Mushrooms=true

Pizza largePep = new Pizza.Builder("Large")

.withPepperoni()

.build();

largePep.display(); // Output: Pizza: Size=Large, Pepperoni=true, Mushrooms=false

// If 'Builder' was non-static, you'd need an OuterClass instance first:

// Pizza outerPizza = new Pizza(...);

// Pizza.Builder builder = outerPizza.new Builder("Medium"); // Incorrect if Builder is static

}

}

**Key Points about static Nested Classes:**

* They can be instantiated directly using the outer class name (OuterClass.NestedClass).
* They **cannot** directly access non-static (instance) fields or methods of the outer class. They can only access static members of the outer class.
* They are functionally similar to top-level classes but are logically grouped within their outer class for organization.

**Important Rules & Best Practices for static**

1. **Access with Class Name:** Always access static members (fields and methods) using the class name (ClassName.staticMember) for clarity. While Java allows objectRef.staticMember, it's misleading.
2. **No this or super in static context:** static methods and static blocks do not operate on an object instance, so they cannot use this (which refers to the current object) or super (which refers to the parent object's methods/fields).
3. **static members live longer:** static members are initialized when the class is loaded and remain in memory for the lifetime of the application (or until the class is unloaded), whereas instance members are created and destroyed with each object.
4. **Overuse is a bad idea:** While powerful, don't overuse static.
   * It promotes tight coupling (harder to change one part without affecting others).
   * It makes testing harder (harder to mock or substitute behavior).
   * It can lead to a less object-oriented design (procedural style).
   * It can lead to concurrency issues if shared static mutable state isn't handled carefully (e.g., using synchronized blocks).

**When to Think Twice About static:**

* If the data/behavior truly belongs to a specific instance of an object.
* If you need polymorphism (static methods cannot be overridden).
* If you need to change the behavior easily (static methods are harder to mock/test).

By understanding the "why, when, and how" of static for fields, methods, initialization blocks, and nested classes, you gain a powerful tool in your Java arsenal. Use it wisely, and only when the class-level nature is truly what you need!

**Why static in main function**

Because to call main function we have to create an object of main first. To avoid this we use static. Because it is static we don’t want to create an object to execute.

If its non static then if we need to call main we have to create an object of its class. If we don’t create an object then how will we call it? Because main is the point of starting an execution, if that’s not started then how will we create an object?

**Encapsulation**

Encapsulation is the mechanism of bundling the data (attributes) and the methods that operate on the data into a single unit, which is called a class. It also involves controlling the access to the internal data of an object and hiding the implementation details from the outside world.

Think of it like a capsule that contains medicine. The medicine (data) is protected within the capsule, and you can only interact with it in a controlled way (through the instructions provided, which are analogous to methods).

**Key Aspects of Encapsulation**:

1. **Data Hiding**: Protecting the internal data of an object from direct access by outside code. This is typically achieved using access modifiers, such as private.
2. **Controlled Access**: Providing controlled access to the data through public methods (getters and setters). These methods act as intermediaries, allowing you to read (get) or modify (set) the data while enforcing certain rules or constraints.

**Benefits of Encapsulation:**

* **Data Protection**: Prevents accidental or unauthorized modification of an object's internal state, ensuring data integrity.
* **Flexibility and Maintainability**: You can change the internal implementation of a class without affecting the code that uses it, as long as the public interface (methods) remains the same. This makes the code more maintainable and easier to evolve.
* **Increased Reusability**: Well-encapsulated classes are more self-contained and easier to reuse in different parts of an application or in other projects.
* **Abstraction**: Encapsulation helps in achieving abstraction by hiding the complex implementation details and exposing only a simplified interface to the user.

**Implementation of Encapsulation in Java:**

1. Declare instance variables as private: This restricts direct access to the attributes from outside the class.
2. Provide public getter methods to access the private data: These methods allow you to retrieve the values of the private attributes. They typically have names like getVariableName().
3. Provide public setter methods to modify the private data (optional): These methods allow you to change the values of the private attributes. They typically have names like setVariableName(dataType newValue). You can include logic within the setter methods to validate the new values before assigning them.

Example: Encapsulating the Car Class

Let's modify our Car class to encapsulate its attributes:

```Java

public class Car {

private String make;

private String model;

private String color;

private int year;

// Constructor

public Car(String make, String model, String color, int year) {

this.make = make;

this.model = model;

this.color = color;

this.year = year;

}

// Public getter methods

public String getMake() {

return make;

}

public String getModel() {

return model;

}

public String getColor() {

return color;

}

public int getYear() {

return year;

}

// Public setter methods (optional - we might not want to allow changing the year, for example)

public void setColor(String newColor) {

this.color = newColor;

}

// Behaviors (methods) - remain public

public void startEngine() {

System.out.println("The " + color + " " + year + " " + make + " " + model + " engine is starting.");

}

public void accelerate() {

System.out.println("The " + model + " is accelerating.");

}

public void displayDetails() {

System.out.println("Make: " + make);

System.out.println("Model: " + model);

System.out.println("Color: " + color);

System.out.println("Year: " + year);

}

public static void main(String[] args) {

Car myCar = new Car("Toyota", "Camry", "Silver", 2022);

// We can no longer directly access myCar.make, myCar.color, etc.

// We must use the getter methods:

System.out.println("Car make: " + myCar.getMake()); // Output: Car make: Toyota

System.out.println("Car color: " + myCar.getColor()); // Output: Car color: Silver

// We can use the setter method to change the color:

myCar.setColor("Blue");

System.out.println("New car color: " + myCar.getColor()); // Output: New car color: Blue

myCar.startEngine(); // The startEngine method still works because it accesses the attributes internally

}

}

```

In this modified Car class:

* The attributes (make, model, color, year) are now declared as private.
* Public getter methods (getMake(), getModel(), getColor(), getYear()) provide read-only access to these attributes.
* A public setter method (setColor()) allows controlled modification of the color attribute. We could choose not to provide setters for other attributes like make or year if we don't want them to be changed after the object is created.

**Constructors:**

**Defining:** public className( ) { }

Constructors are used to create objects and gives initial value.

A constructor never returns anything.

A constructor is a special method with same name as class name.

Constructors are used to set the default values to instance variables

If you don’t set the constructor, it will automatically define a default constructor with empty values.

By encapsulating the data, we protect the internal state of the Car object and provide a controlled way to interact with it. This makes the class more robust and easier to maintain.

```Java

class Student

{

string name;

int age;

// Default constructor

public Student() {

name = “John”;

age = 18;

}

// Parameterised constructor

public Student(String name, int age) {

this.name = name;

this.age = age;

}

}

```

**Super**

Every constructor has its first statement super();

You mention it or not it will be there.

The super keyword is used in a subclass to:

* Call the constructor of the superclass. This must be the first statement in the subclass's constructor. If you don't explicitly call super(), Java will implicitly call the no-argument constructor of the superclass. If the superclass doesn't have a no-argument constructor, you must explicitly call one of the superclass's constructors using super(arguments).
* Access members (methods and attributes) of the superclass that have been hidden by members with the same name in the subclass.

```Java

class A

{

// Default constructor

public A() {

System.out.println(“In A”);

}

// Parameterized super

public A(int n) {

System.out.println(“In A int”);

}

}

class B extends A

{

// Default constructor

public B() {

System.out.println(“In A”);

}

// To call the parameterised constructor of the super class you have to pass the parameter while calling the constructor.

public B(int n) {

super(n)

System.out.println(“In A”);

}

// this will call the default constructor B and after super.

public B(int n) {

this()

System.out.println(“In A”);

}

}

Public static void main(String[] args) {

B obj = new B() // this will call the default constructor of super class A’s constructor first.

B obj2 = new B(n) // this will call the parametrised constructor of super class A’s constructor first.

}

```

**Every class in java extends the Object class.**

That’s why we have super in class A even though there is no super class above class A in the above code.

**Anonymous object**

We cannot reuse anonymous objects.

Every time you use it will create a new object.

```Java

Clas Anonymous

{

Public void show() { System.out.println(“Anonymous Object”);

}

Class Demo

{

Public static void main (String[] args) {

Anonymous obj = new Anonymous();

// instead of the above we can use the below

new Anonymous();

new Anonymous().show;

}

}

```

**Inheritance**

We have single level inheritance and multi level inheritance.

**Inheritance** is a mechanism in which a new class (called a **subclass** or **derived class**) inherits properties (attributes) and behaviors (methods) from an existing class (called a **superclass** or **base class** or **parent class**). Inheritance establishes an "is-a" relationship between the subclass and the superclass. For example, a Dog is a type of Animal.

**Key Concepts in Inheritance:**

* **Superclass (Base Class or Parent Class):** The class whose properties and behaviors are inherited.
* **Subclass (Derived Class or Child Class):** The class that inherits from the superclass. The subclass can add its own unique properties and behaviors or override the inherited ones.
* **extends Keyword:** In Java, the extends keyword is used to indicate that a class is inheriting from another class.
* **Code Reusability:** Inheritance promotes code reusability by allowing subclasses to reuse the code defined in their superclass.
* **Method Overriding:** A subclass can provide its own implementation of a method that is already defined in its superclass. This allows a subclass to modify or extend the behavior inherited from the superclass.

**Syntax for Inheritance in Java:**

```Java

class SubclassName extends SuperclassName {

// Attributes and methods specific to the subclass

// or overridden methods from the superclass

}

```

**The super Keyword:** see the last Super keyword topic

**Example: Animal and Dog Classes**

Let's create a simple example with an Animal superclass and a Dog subclass:

```Java

// Superclass: Animal

class Animal {

String name;

public Animal(String name) {

this.name = name;

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

}

public void eat() {

System.out.println(name + " is eating.");

}

public void makeSound() {

System.out.println("Animal makes a generic sound.");

}

}

// Subclass: Dog (inherits from Animal)

class Dog extends Animal {

String breed;

public Dog(String name, String breed) {

super(name); // Call the constructor of the Animal superclass

this.breed = breed;

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

}

// Override the makeSound method

@Override

public void makeSound() {

System.out.println(name + " (a " + breed + ") barks: Woof!");

}

public void fetch() {

System.out.println(name + " is fetching the ball.");

}

public static void main(String[] args) {

Dog myDog = new Dog("Buddy", "Golden Retriever");

myDog.eat(); // Inherited from Animal

myDog.makeSound(); // Overridden method in Dog

myDog.fetch(); // Specific to Dog

Animal genericAnimal = new Animal("Generic Animal");

genericAnimal.eat();

genericAnimal.makeSound();

}

}

```

In this example:

* Animal is the superclass with attributes (name) and methods (eat(), makeSound()).
* Dog is the subclass that extends Animal. It inherits the name attribute and the eat() method.
* The Dog class has its own additional attribute (breed) and a specific method (fetch()).
* The makeSound() method is **overridden** in the Dog class to provide a more specific behavior for a dog. The @Override annotation is a good practice to indicate that a method is intended to override a superclass method; the compiler can then check if the override is valid.
* In the Dog constructor, super(name) is used to call the constructor of the Animal superclass to initialize the inherited name attribute. This must be the first statement in the Dog constructor.

**Types of Inheritance in Java:**

Java supports **single inheritance** for classes, meaning a class can directly inherit from only one superclass. However, Java supports **multiple inheritance of interfaces** (a class can implement multiple interfaces), which we will discuss later under Abstraction and Interfaces.

**Benefits of Inheritance:**

* **Code Reusability:** Subclasses inherit the attributes and methods of the superclass, reducing the need to write the same code multiple times.
* **Maintainability:** Changes made in the superclass are automatically reflected in all its subclasses (unless overridden), making maintenance easier.
* **Extensibility:** You can extend the functionality of existing classes by creating new subclasses that add specific behaviors.
* **Polymorphism (as we'll see next):** Inheritance forms the basis for polymorphism, allowing objects of different classes to be treated as objects of a common superclass type.

Inheritance is a powerful tool in OOP that helps in organizing code into a hierarchy and promoting reusability.

**Packages**

Packages are folders.

We can put multiple folders and files in a single folder.

To get the access to other files in our current fule use “import” to call it.

**Access Modifiers**

As the name suggests this defines the accessibility of the variable, methods in a class for other classes.

**public :** can be accessed from anywhere

**private :** variables or methods defined as private can only be accessed from within the same class.

**Default :** can be accessed from with in a package. It is the default access if you you don’t mention any modifiers specifically.

**Protected :** cannot acces from out side the package, but it can be accessed if that is a sub-class.

Note :

A file can have only one class.

Make the instance variables private.

Make the methods public.

Try not to use default.

**Polymorphism**

**Polymorphism** (which literally means "many forms") is the ability of an object to take on many forms. In Java, polymorphism is primarily achieved through two mechanisms:

1. Method Overloading (Compile-time Polymorphism or Static Polymorphism)
2. Method Overriding (Runtime Polymorphism or Dynamic Polymorphism)

Let's explore each of these in detail.

**1. Method Overloading (Compile-time Polymorphism):**

Method overloading occurs when a class has multiple methods with the same name but different **parameter lists**. The parameter lists must differ in either the number of parameters, the data types of the parameters, or the order of the parameters. The return type of the methods can be the same or different, but it's not sufficient to distinguish overloaded methods.

The compiler determines which overloaded method to call based on the arguments passed in the method call. This decision is made at compile time, hence the name "compile-time polymorphism."

**Example of Method Overloading:**

```Java

class Calculator {

public int add(int a, int b) {

System.out.println("Adding two integers");

return a + b;

}

public double add(double a, double b) {

System.out.println("Adding two doubles");

return a + b;

}

public int add(int a, int b, int c) {

System.out.println("Adding three integers");

return a + b + c;

}

public int add(int a, double b) {

System.out.println("Adding an integer and a double");

return a + (int) b; // Casting double to int

}

public double add(double a, int b) {

System.out.println("Adding a double and an integer");

return a + b;

}

public static void main(String[] args) {

Calculator calc = new Calculator();

System.out.println(calc.add(5, 3)); // Calls add(int, int)

System.out.println(calc.add(2.5, 3.5)); // Calls add(double, double)

System.out.println(calc.add(1, 2, 3)); // Calls add(int, int, int)

System.out.println(calc.add(10, 2.7)); // Calls add(int, double)

System.out.println(calc.add(3.14, 5)); // Calls add(double, int)

}

}

```

In this Calculator class, the add method is overloaded multiple times with different parameter lists. When you call the add method, the Java compiler looks at the arguments you provide and matches them to the appropriate method signature.

**2. Method Overriding (Runtime Polymorphism):**

Method overriding occurs when a subclass provides a specific implementation for a method that is already defined in its superclass. The overridden method in the subclass must have the same name, the same parameter list, and the same return type (or a covariant return type, which is a subclass of the superclass's return type) as the method in the superclass.

Runtime polymorphism is achieved through **late binding** or **dynamic method dispatch**. The decision of which method to call is made at runtime based on the actual type of the object being referred to, not the reference type. This typically involves inheritance and using a superclass reference to refer to a subclass object.

**Example of Method Overriding (Revisiting Animal and Dog):**

```java

class Animal { public void makeSound() { System.out.println("Animal makes a generic sound."); } }

class Dog extends Animal { @Override public void makeSound() { System.out.println("Dog barks: Woof!"); } }

class Cat extends Animal { @Override public void makeSound() { System.out.println("Cat meows: Meow!"); } }

public class PolymorphismDemo { public static void main(String[] args) { Animal animal1 = new Animal(); Animal animal2 = new Dog(); // Superclass reference to a subclass object Animal animal3 = new Cat(); // Superclass reference to a subclass object

animal1.makeSound(); // Output: Animal makes a generic sound. (Animal type)

animal2.makeSound(); // Output: Dog barks: Woof! (Dog type at runtime)

animal3.makeSound(); // Output: Cat meows: Meow! (Cat type at runtime)

// You can also have an array of Animal references pointing to different animal objects

Animal[] animals = new Animal[3];

animals[0] = new Animal();

animals[1] = new Dog();

animals[2] = new Cat();

for (Animal animal : animals) {

animal.makeSound();

/\*

Output:

Animal makes a generic sound.

Dog barks: Woof!

Cat meows: Meow!

\*/

}

}

}

```

In this example:

\* The `Animal` class has a `makeSound()` method.

\* The `Dog` and `Cat` classes inherit from `Animal` and \*\*override\*\* the `makeSound()` method to provide their specific sounds.

\* In the `main` method, we see runtime polymorphism in action. When an `Animal` reference (`animal2`, `animal3`, and the elements of the `animals` array) points to a `Dog` or `Cat` object, the overridden `makeSound()` method of the actual object type (Dog or Cat) is called at runtime.

**Key Differences between Overloading and Overriding:**

| Feature | Method Overloading | Method Overriding |

| :--------------- | :------------------------------------- | :-------------------------------------- |

| \*\*Scope\*\* | Within the same class | In a subclass of the superclass |

| \*\*Parameter List\*\* | Must be different (number, type, order) | Must be the same |

| \*\*Return Type\*\* | Can be the same or different | Must be the same (or covariant) |

| \*\*Access Modifiers\*\* | Can be different | Cannot be more restrictive than the superclass |

| \*\*Binding\*\* | Compile-time (static binding) | Runtime (dynamic binding) |

| \*\*Purpose\*\* | To provide multiple ways to call a method | To provide a specific implementation in a subclass |

Polymorphism is a powerful concept that allows for more flexible and extensible code. It enables you to write code that can work with objects of different classes in a uniform way, as long as they share a common superclass or implement the same interface…

**Dynamic Method Dispatch**

```Java

Class A {

Public void show( ) {

System.out.println(“In a show”);

}

}

Class B extends A {

Public void show( ) {

System.out.println(“In B show”);

}

}

Class C extends A {

Public void show( ) {

System.out.println(“In C show”);

}

}

public Class Main {

Public static void main( ) {

// Normal way

B obj = new B();

obj.show();

// Dynamic way way

// Type can be of Parrent but implementation is of sub class.

A obj = new B();

obj.show(); // result will be in B show

obj =new C();

obj.show(); // result will be in C show

}

}

```

This is called Dynamic method dispatch. Which shows runtime polymorphism in action. Different object will have behaviour.

Irrelevant of what type variable you have its up to what object is being implemented.

**Final Keyword**

Can be used with variables and classes .

* It is used to make immutable varibles. Once declared and initialized it cannod be changed.
* Used with class it will stop the possibility of inheritance.
* Used with methods it stops method overloading

**Object Class**

We learned every class in java extend an object class.

So when we try to print an obj it will give us an output something like this: ClassName@7ad042f3.

To override this create a method called toString, which is available in the parent Object class.

```Java

Class Laptop

{

String model;

int price;

// Overrides the Object method toString() which converts the whole value to a hash code

public String toString() {

return model + “ : “ price;

}

// checks the equality of two object. Overriding the parent method.

// these with IDE provided method creation.

public boolean equals(Laptop that) {

if (this.model.equals(that.model) && this.price == that.price)

{

return true;

}

else

return false;

}

}

```

**Down Casting and Up Casting**

``` Java

class A {

public void show1() {

System.out.print (“ In A show”);

}

}

class B extends A {

public void show2() {

System.out.print (“ In B show”);

}

}

public Class Main {

Public static void main( ) {

// Type casting

double d = = 4.5;

int i = (int)d;

System.out.print(i);

// Type casting in OOPs

// This is upcasting

A obj = (A) new B();

// Down casting

We are down grading the obj of parent type to subclass type.

// Then onlywe will be able to call show2() in B class

B obj1 = obj; // wrong will not be able to call show2().

B obj = (B) obj; // right way, down grading the obj of parent type to chld class.

Obj.show2();

}

}

```

**Wrapper Class**

Primitive dta type does not extends object class in java. That’s why they are stored in stack memory and not heap.

For every primitive type we are going to have a class for it.

For int we have Integer class which extends the object class.

For char we have Character class which extends the object class.

For double we have Double class which extends the object class.

For int we have Integer class which extends the object class.

**Boxing and auto-boxing Concept**

```Java

public class Main {

public static void main(String[] args) {

int num = 7;

Integer num1 = new Integer(num);

// Deprecated. This is Boxing, taking a primitive value and wrapping it in class object.

// New way

Integer num1 = num;

// here the conversion happens automatically so it is called auto boxing

int num2 = num1.intValue();

// converting an object to primitive type, unboxing

int num2 = num1;

// converting an object to primitive type happening automatically so unboxing

String str = “7”;

Int num3 = Integer.parseInt(str);

// converting String to integer. Integer is a wrapper class that has multiple methods.

}

}

```

**Abstract Keyword**

Abstraction is where you declare the methods not the implementations.

To only declare a method, we use abstract keyword.

An abstract method can only be created in an abstract class

If you are extending an abstract class then you must implement it els it will show error.

We cannot create the object of abstract class.

Abstract class can have normal concrete methods and abstract methods.

```Java

abstract class Car {

Public abstract void drive();

// This declaration. Implementation happens in the derived class

Public void playMusic() {

System.out.println(“Playing music…”);

}

// here declaration and implementation is happening

}

Class Innova extends Car {

Public void drive() {

System.out.println(“Driving….”);

}

// Implementation of abstract method in parent class

}

Public class Main {

Public static void main(String[] args) {

Car obj = new Car(); // wrong, cannot create obj of abstract class

Innova Car = new Innova(); // this is allowed

Car.drive(); // result: Driving….

}

}

```

**Inner Class**

This is class inside another class.

```Java

class A {

int num = 8;

public void show() {

System.out.println(“Inside A show”);

}

Class B {

public void innerClassMethod() {

System.out.println(“ner class method..”);

}

}

}

public class Main {

public static void main() {

A obj = new A();

Obj.show();

// To call a method inside an inner class you must first create a reference to the parent, the using the parent you need to create new. Then only it will work

A.B obj1 = obj.new B);

Obj1. innerClassMethod();

// ifyou want to call like below the make inner class static.

A>B obj2 = new A>B();

obj2.innerClassMethod();

}

}

```

**Anonymous Inner Class**

```Java

class A {

public void show() {

System.out.println(“Inside A show”);

}

}

Class B extend A {

public void show() {

System.out.println(“Inside A show”);

}

}

public class Main {

public static void main() {

A obj = new A() {

public void show() {

System.out.println(“Inside A show”);

}

}

}

}

```

If you want to run the abstract class and abstract method once then use anonymous inner class to implement it.

```Java

abstract class A {

public abstract void show() ;

}

public class Main {

public static void main() {

A obj = new A() {

public void show() {

System.out.println(“Inside A show”);

}

}

}

}

```

**Enums**

Enums are like class with objects

These object in enums are named constants.

``Java

Enum Status {

Running, Failed, Pending, Success;

}

Public class Main {

Public static main(String[] args) {

Status s = Status.Running;

//To print all status

Status[] ss = Status.values();

for(Status s : ss) {

System.out.println(s);

// to print the order

System.out.println(s.ordinal());

}

}

}

```

**Enum with if els and switch**

``Java

Enum Status {

Running, Failed, Pending, Success;

}

Public class Main {

Public static main(String[] args) {

Status s = Status.Running;

If(s == Status.Running)

System.out.println(“All Good”);

else if (s == Status.Failed)

System.out.println(“Try Again”);

else if (s == Status.Pending)

System.out.println(“Please Wait”);

else

System.out.println(“Done”);

}

}

}

```

``Java

Enum Status {

Running, Failed, Pending, Success;

}

Public class Main {

Public static main(String[] args) {

Status s = Status.Running;

// Using Switch

switch(s)

{

case Running:

System.out.println(“All Good”);

break;

case Failed:

System.out.println(“Try again”);

break;

case Pending:

System.out.println(“Please wait”);

break;

default:

System.out.println(“Done”);

break;

}

}

}

}

```

Enums does not supports extend.

Enum in java is a class that extends parent class enum

But you can declare variables, methods, constructors

**Constructors with Enum**

``Java

Enum Laptop {

Macbook(price: 2000), XPS(price: 2500), Surface(price: 1800),

private int price;

private Laptop(int price) {

this.price = price

}

Public int getPrice() {

return price;

}

Public int setPrice() {

return price;

}

}

Public class Main {

Public static main(String[] args) {

Laptop lap = Laptop.Macbook;;

System.out.println(lap +” : ” + lap.getPrice());

for(Laptop lap: Laptop.values()) {

System.out.println(lap + “ : “ + price);

}

}

}

```

If one constant in the above example is without price along with it the you have to create a default constructor without parameter and define the price there or else it will show error.

**Sealed Classes**

Let's dive into **Sealed Classes in Java**, a feature introduced as a preview in Java 15, finalized in Java 17. Sealed classes and interfaces provide a way to restrict which other classes or interfaces can extend or implement them. This allows you to define a controlled hierarchy of types, enhancing both expressiveness and maintainability, especially when working with pattern matching.

**What are Sealed Classes?**

A **sealed class** (or interface) is a class (or interface) that explicitly lists the classes that are permitted to directly extend it. This means you can control the entire set of possible subclasses for a given sealed type. If a class or interface is not explicitly listed as permitted, it cannot extend or implement the sealed type.

**Why were Sealed Classes Introduced?**

Before sealed classes, if you wanted to restrict subclassing, your options were:

1. **final keyword:** This prevents any subclassing at all.
2. **Package-private (default) access:** This restricts subclassing to within the same package, but not to specific classes.
3. **Constructors with restricted visibility:** You could make constructors private or protected, but this doesn't fully prevent internal classes or nested classes from extending.

None of these options provided the precise control needed for scenarios where you want to allow *some* specific subclasses but disallow *all others*. Sealed classes fill this gap, enabling exhaustive pattern matching (e.g., with switch expressions) and providing a clear definition of a closed hierarchy.

**Syntax of Sealed Classes:**

To declare a sealed class, you use the sealed keyword along with the permits keyword, followed by a comma-separated list of classes that are allowed to extend it.

```Java

public sealed class Shape permits Circle, Square, Triangle {

// ...

}

```

**Rules for Permitted Subclasses:**

When a class or interface A is declared sealed and permits B, C, D:

1. **Direct Subclasses Only:** Only B, C, and D are allowed to directly extend A. No other class can extend A.
2. **Must be in the Same Module (or Package):** Permitted subclasses must be in the same module as the sealed class. If they are in the unnamed module (the typical case for most applications), they must be in the same package. This constraint ensures that the compiler can easily verify the complete hierarchy.
3. **Permitted Subclasses Must Declare Their Sealing Status:** Each permitted subclass must explicitly declare how it continues the sealing. It can be declared as:
   * **final:** Prevents any further subclassing. This effectively terminates the hierarchy for that branch.
   * **sealed:** Allows further subclassing, but again, only by a specified set of permitted subclasses.
   * **non-sealed:** Allows any class to extend it. This opens up the hierarchy for that branch.

**Example 1: Basic Sealed Class Hierarchy**

Let's define a Shape sealed class and some permitted subclasses.

```Java

// Shape.java

public sealed class Shape permits Circle, Square, Triangle {

public abstract String getType();

}

// Circle.java

public final class Circle extends Shape {

private double radius;

public Circle(double radius) {

this.radius = radius;

}

public double getRadius() {

return radius;

}

@Override

public String getType() {

return "Circle";

}

}

// Square.java

public non-sealed class Square extends Shape {

private double side;

public Square(double side) {

this.side = side;

}

public double getSide() {

return side;

}

@Override

public String getType() {

return "Square";

}

}

// Triangle.java

public sealed class Triangle extends Shape permits EquilateralTriangle, IsoscelesTriangle {

private double side1, side2, side3;

public Triangle(double side1, double side2, double side3) {

this.side1 = side1;

this.side2 = side2;

this.side3 = side3;

}

@Override

public String getType() {

return "Triangle";

}

}

// EquilateralTriangle.java

public final class EquilateralTriangle extends Triangle {

public EquilateralTriangle(double side) {

super(side, side, side);

}

@Override

public String getType() {

return "Equilateral Triangle";

}

}

// IsoscelesTriangle.java

public final class IsoscelesTriangle extends Triangle {

public IsoscelesTriangle(double base, double leg) {

super(base, leg, leg);

}

@Override

public String getType() {

return "Isosceles Triangle";

}

}

```

In this example:

* Shape is sealed and permits Circle, Square, and Triangle.
* Circle is final, meaning no class can extend Circle.
* Square is non-sealed, meaning any class can extend Square.
* Triangle is sealed again, restricting its direct subclasses to EquilateralTriangle and IsoscelesTriangle.
* EquilateralTriangle and IsoscelesTriangle are final.

**What if you try to extend a sealed class without permission?**

```Java

// InvalidShape.java - This will cause a compile-time error

// error: class IllegalShape is not allowed to extend sealed class Shape

public class IllegalShape extends Shape {

@Override

public String getType() {

return "Illegal";

}

}

```

The compiler will prevent IllegalShape from extending Shape because IllegalShape is not listed in Shape's permits clause.

**Sealed Interfaces:**

Sealed interfaces work similarly to sealed classes. They restrict which classes can implement them or which interfaces can extend them.

```Java

public sealed interface Service permits UserService, ProductService {

String getServiceName();

}

public final class UserService implements Service {

@Override

public String getServiceName() {

return "User Service";

}

}

public non-sealed class ProductService implements Service {

@Override

public String getServiceName() {

return "Product Service";

}

}

// AnotherService.java - This will cause a compile-time error

// error: class AnotherService is not allowed to implement sealed interface Service

public class AnotherService implements Service {

@Override

public String getServiceName() {

return "Another Service";

}

}

```

**Benefits of Sealed Classes and Interfaces:**

1. **Exhaustive Pattern Matching (with switch expressions/statements):** This is one of the most powerful benefits. When you use switch expressions or switch statements with instanceof patterns (introduced in Java 17), the compiler can determine if you've covered all possible permitted subclasses of a sealed type. If you miss a case, the compiler will give you a warning or an error, ensuring exhaustiveness.

```Java

public class ShapeProcessor {

public static double calculateArea(Shape shape) {

return switch (shape) {

case Circle c -> Math.PI \* c.getRadius() \* c.getRadius();

case Square s -> s.getSide() \* s.getSide();

case Triangle t -> { // Assuming you have a method to calculate triangle area

// For simplicity, let's just return 0 for now

yield 0.0;

}

// No 'default' needed because the compiler knows all permitted subclasses are covered

// If you miss a case for a permitted subclass, the compiler will warn/error

};

}

public static void main(String[] args) {

Shape circle = new Circle(5);

Shape square = new Square(4);

Shape equiTriangle = new EquilateralTriangle(6);

System.out.println("Area of Circle: " + calculateArea(circle)); // Output: Area of Circle: 78.53981633974483

System.out.println("Area of Square: " + calculateArea(square)); // Output: Area of Square: 16.0

System.out.println("Area of Equilateral Triangle: " + calculateArea(equiTriangle)); // Output: Area of Equilateral Triangle: 0.0

}

}

```

If calculateArea method above was missing the Triangle case, the compiler would issue a warning like: warning: the switch expression does not cover all possible input values (use a default instance label). This is a huge benefit for writing robust and type-safe code.

1. **Improved Code Clarity and Maintainability:** The permits clause clearly documents the intended hierarchy, making it easier for developers to understand the design and constraints of a class or interface. This reduces the chance of unintended extensions.
2. **Enhanced Security (to some extent):** By limiting extensibility, you can prevent malicious or unintended subclasses from interfering with the internal workings of your sealed type.
3. **Better API Design:** Sealed types allow you to design APIs where certain types are known to be part of a closed set, making the API more predictable and easier to work with.

**Limitations and Considerations:**

* **Same Module/Package Constraint:** Permitted subclasses must be in the same module as the sealed class (or same package if in the unnamed module). This can be a limiting factor for very large projects that are split across many modules.
* **Compile-Time Check:** The sealing is enforced at compile time.
* **Verbosity for Complex Hierarchies:** For very deep or wide hierarchies, listing all permitted subclasses can become verbose.
* **Backward Compatibility:** Code compiled with older Java versions won't understand sealed classes.

**When to Use Sealed Classes:**

Sealed classes are particularly useful in scenarios where you have:

* **Fixed sets of types:** E.g., different types of Shape, Result (Success, Failure), Command objects in a command pattern.
* **When using pattern matching:** To enable exhaustive switch expressions that the compiler can verify.
* **When defining algebraic data types:** Where a type can be one of a fixed number of possibilities.
* **Internal Frameworks/APIs:** To control how consumers extend your core types.

**Comparison with Enums:**

While enums also represent a fixed set of constants, sealed classes are more flexible because:

* **They can be classes:** Enums are specialized classes themselves. Sealed classes allow for more complex state and behavior in each permitted subclass.
* **They can have different types:** Each permitted subclass can be a distinct type with its own unique properties and methods, whereas enum constants are all of the same enum type.
* **They can be extended (with sealed or non-sealed):** Enums cannot be extended.

**Conclusion:**

Sealed classes and interfaces in Java 17+ are a powerful addition that provides fine-grained control over inheritance hierarchies. They enhance code clarity, improve maintainability, and, most importantly, enable exhaustive pattern matching with switch expressions, leading to more robust and type-safe code. By explicitly declaring the permitted subclasses, you can build more predictable and well-defined type systems in your Java applications.

**Record Classes**

Let's delve into **Record Classes in Java**, a significant feature introduced as a preview in Java 14 and finalized in Java 16. Records are a special kind of class designed to be concise and immutable carriers of data. They address a common pain point in Java: the boilerplate code traditionally required for simple data-carrying classes.

**What are Record Classes?**

A **record class** (or simply "record") is a new type of class in Java specifically designed to model plain data aggregates. Its primary purpose is to hold data. When you declare a record, the Java compiler automatically generates a lot of boilerplate code that you would typically write manually for a data-centric class, such as:

1. **A canonical constructor:** A constructor that takes all the components of the record as arguments.
2. **Accessor methods:** A public accessor method for each component (e.g., name() for a name component). These methods follow the name of the component, not the traditional get prefix.
3. **equals() and hashCode() methods:** Implementations that consider all components for equality and hashing.
4. **toString() method:** A useful string representation of the record's components.

Records are implicitly final, meaning they cannot be extended. They also implicitly extend java.lang.Record, which is the base class for all records.

**Why were Record Classes Introduced?**

Before records, creating a simple data class in Java required a significant amount of boilerplate code:

```Java

// Traditional Java Data Class

public class Point {

private final int x;

private final int y;

public Point(int x, int y) {

this.x = x;

this.y = y;

}

public int getX() {

return x;

}

public int getY() {

return y;

}

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Point point = (Point) o;

return x == point.x && y == point.y;

}

@Override

public int hashCode() {

return Objects.hash(x, y);

}

@Override

public String toString() {

return "Point{" +

"x=" + x +

", y=" + y +

'}';

}

}

```

This is a lot of code for a simple data holder. Records dramatically reduce this verbosity, making the code cleaner, more readable, and less prone to errors when changes are made.

**Syntax of Record Classes:**

You declare a record using the record keyword, followed by the record name and a parenthesized list of its components (the "record header").

```Java

public record Point(int x, int y) { }

```

That's it! This single line generates all the boilerplate methods mentioned above.

**Example 1: Simple Record**

```Java

// User.java

public record User(int id, String name, String email) {

// Records are implicitly final, so you can't extend them.

// Records implicitly implement equals(), hashCode(), and toString().

// They also have accessor methods named after the components (e.g., id(), name(), email()).

}

public class RecordExample {

public static void main(String[] args) {

User user1 = new User(1, "Alice", "alice@example.com");

User user2 = new User(2, "Bob", "bob@example.com");

User user3 = new User(1, "Alice", "alice@example.com"); // Same data as user1

// Accessor methods

System.out.println("User 1 ID: " + user1.id());

System.out.println("User 1 Name: " + user1.name());

// toString() method

System.out.println("User 1: " + user1); // Output: User[id=1, name=Alice, email=alice@example.com]

System.out.println("User 2: " + user2); // Output: User[id=2, name=Bob, email=bob@example.com]

// equals() method

System.out.println("User 1 equals User 3: " + user1.equals(user3)); // Output: User 1 equals User 3: true

System.out.println("User 1 equals User 2: " + user1.equals(user2)); // Output: User 1 equals User 2: false

// hashCode() method

System.out.println("User 1 hash code: " + user1.hashCode());

System.out.println("User 3 hash code: " + user3.hashCode()); // Same as user1.hashCode()

}

}

```

**Customizations in Records:**

While records aim for conciseness, you can still add custom behavior:

1. **Compact Constructors:** You can define a more compact version of the canonical constructor if you need to perform validation or normalization on the components. The parameters are implicitly declared, and the assignments are automatically performed *after* the compact constructor body.

```Java

public record Range(int min, int max) {

// Compact constructor for validation

public Range { // No parameters needed, they are implicit

if (min > max) {

throw new IllegalArgumentException("min cannot be greater than max");

}

// min = min; max = max; -- these assignments are implicitly done after this block

}

}

public class CompactConstructorExample {

public static void main(String[] args) {

Range validRange = new Range(10, 20);

System.out.println("Valid Range: " + validRange); // Output: Valid Range: Range[min=10, max=20]

try {

Range invalidRange = new Range(20, 10); // This will throw an exception

System.out.println("Invalid Range: " + invalidRange);

} catch (IllegalArgumentException e) {

System.out.println("Error: " + e.getMessage()); // Output: Error: min cannot be greater than max

}

}

}

```

1. **Custom Canonical Constructor:** If you need to perform more complex logic *before* the component assignments, or if you want to explicitly declare the parameters, you can write a full canonical constructor. However, this largely negates the conciseness benefit.

```Java

public record Product(String name, double price) {

public Product(String name, double price) {

if (name == null || name.isBlank()) {

throw new IllegalArgumentException("Name cannot be empty");

}

if (price < 0) {

throw new IllegalArgumentException("Price cannot be negative");

}

this.name = name.trim(); // Normalization

this.price = price;

}

}

```

Note that the this.name = name; this.price = price; assignments *must* be done explicitly if you write a full canonical constructor. The compact constructor is preferred for most validation/normalization tasks.

1. **Additional Fields and Methods:** You can add static fields, static methods, instance methods, and even implement interfaces in a record.

```Java

public interface HasDescription {

String getDescription();

}

public record Item(String name, double weight) implements HasDescription {

// Static field

public static final String DEFAULT\_UNIT = "kg";

// Instance method

public double calculateCost(double pricePerUnit) {

return weight \* pricePerUnit;

}

// Implementation of interface method

@Override

public String getDescription() {

return "Item: " + name + " (" + weight + Item.DEFAULT\_UNIT + ")";

}

// Static method

public static Item createLightItem(String name) {

return new Item(name, 0.1);

}

}

public class CustomRecordExample {

public static void main(String[] args) {

Item book = new Item("Java Basics", 1.2);

System.out.println(book.getDescription()); // Output: Item: Java Basics (1.2kg)

System.out.println("Cost of book: " + book.calculateCost(10.0)); // Output: Cost of book: 12.0

Item feather = Item.createLightItem("Feather");

System.out.println(feather.getDescription()); // Output: Item: Feather (0.1kg)

}

}

```

**Restrictions and Limitations of Records:**

* **Implicitly final:** Records cannot be extended by other classes. This is fundamental to their role as simple data carriers.
* **Cannot extend other classes:** A record implicitly extends java.lang.Record, so it cannot explicitly extend any other class.
* **Cannot declare instance fields outside the record header:** All instance fields are declared as components in the record header. You cannot add additional instance fields directly in the record body.
* **Cannot declare native methods:** (Rarely used anyway in typical application development).
* **No explicit set methods:** By design, records are immutable. Their accessor methods (componentName()) only provide read access. If you need to "modify" a record, you create a new record with the desired changes (similar to how String works).

**When to Use Records:**

Records are ideal for scenarios where you need:

* **Plain Data Transfer Objects (DTOs):** For carrying data between layers of an application.
* **Value Objects:** Objects whose equality is based on the values of their components (e.g., Point, Money, Coordinates).
* **Return types from methods:** Especially when returning multiple values.
* **Keys in Maps:** Since equals() and hashCode() are well-defined.
* **Temporary Data Structures:** For short-lived data.
* **Pattern Matching (with switch expressions/statements):** Records integrate very well with pattern matching (introduced in Java 17+).

```Java

// Assuming records Point and Circle (from earlier discussion of sealed classes)

public record Point(int x, int y) {}

public record Circle(Point center, int radius) {}

public static void printShapeInfo(Object obj) {

if (obj instanceof Point(int x, int y)) { // Record pattern matching (Java 16+)

System.out.println("It's a Point at (" + x + ", " + y + ")");

} else if (obj instanceof Circle(Point p, int r)) {

System.out.println("It's a Circle centered at (" + p.x() + ", " + p.y() + ") with radius " + r);

}

}

public static void main(String[] args) {

printShapeInfo(new Point(10, 20));

printShapeInfo(new Circle(new Point(0, 0), 5));

}

```

**Records vs. Lombok:**

Tools like Project Lombok have long been used to reduce boilerplate code by generating getters, setters, equals, hashCode, and toString methods at compile time. While Lombok is very popular, records offer a native, standard Java solution for data classes.

* **Native vs. Third-Party:** Records are a built-in Java language feature; Lombok is a third-party annotation processor.
* **Immutability:** Records are immutable by default and design, which is a significant advantage for concurrent and predictable code. Lombok allows for mutable data classes unless configured otherwise.
* **Readability:** Records are often considered more readable as the core intent (it's a data class) is immediately clear from the record keyword.
* **Future Enhancements:** Records integrate with other newer Java features like pattern matching.

For simple, immutable data carriers, records are generally preferred over Lombok due to their native support and inherent immutability. For more complex classes with mutable state or custom logic beyond basic data access, traditional classes (potentially with Lombok) are still appropriate.

**Conclusion:**

Record classes are a powerful and welcome addition to Java, significantly reducing the boilerplate code for data-centric classes. They promote immutability, enhance code readability, and integrate seamlessly with newer Java features like pattern matching. By understanding when and how to use records, you can write cleaner, more robust, and more expressive Java code.

**Annotations**

Annotations, introduced in Java 5, provide a way to add metadata to your code. This metadata can be used by the compiler or by other tools and libraries to perform various actions or checks.

**What are Annotations?**

Annotations are a form of syntactic metadata that can be embedded in Java source code. They provide data about the code that is not part of the program's logic itself. Annotations do not directly affect the execution of the code they annotate, but they can be read and processed by the compiler, JVM, or other annotation processing tools.

Think of annotations as labels or tags that you can put on different parts of your code (classes, methods, variables, parameters, etc.) to provide extra information about them.

**Basic Syntax of Annotations:**

Annotations have a special syntax that starts with an @ symbol, followed by the annotation name and optionally, parameters enclosed in parentheses.

```Java

@AnnotationName

@AnnotationName(parameter1 = value1)

@AnnotationName("value") // If the annotation has a single parameter named "value"

@AnnotationName(value = "someValue", other = 123)

```

**Types of Annotations:**

There are three main categories of annotations based on their availability:

1. **Source-level Annotations:** These are retained only in the source code and are discarded by the compiler. They are typically used by static analysis tools.
2. **Class-level Annotations:** These are stored in the .class file but are not available at runtime by default. They can be made available at runtime using the @Retention(RetentionPolicy.RUNTIME) meta-annotation.
3. **Runtime Annotations:** These are stored in the .class file and are available to the JVM at runtime. They can be accessed and processed by the program using reflection.

**Built-in Annotations in Java:**

Java provides several built-in annotations that are commonly used:

* **@Override:** Used to indicate that a method in a subclass is intended to override a method in its superclass. The compiler checks if the superclass actually has a method with the same signature, and it will produce an error if it doesn't. This helps prevent accidental typos in method names.

```Java

class Animal {

public void makeSound() {

System.out.println("Generic animal sound");

}

}

class Dog extends Animal {

@Override

public void makeSound() {

System.out.println("Woof!");

}

}

```

* **@Deprecated:** Used to mark a class, method, or field as obsolete and should no longer be used. Compilers often issue warnings when code uses deprecated elements. The annotation can also include a comment about why it's deprecated and what to use instead.

```Java

class OldClass {

@Deprecated(since = "1.5", forRemoval = true)

public void oldMethod() {

System.out.println("This method is deprecated.");

}

public void newMethod() {

System.out.println("Use this new method instead.");

}

}

```

* **@SuppressWarnings:** Used to suppress compiler warnings that might arise from certain code constructs. You can specify the types of warnings to suppress (e.g., "unchecked", "deprecation", "rawtypes"). Use this annotation judiciously, as it can hide potential issues.

```Java

import java.util.ArrayList;

import java.util.List;

public class WarningDemo {

@SuppressWarnings("unchecked")

public static void main(String[] args) {

List rawList = new ArrayList(); // Creates a raw type list

rawList.add("Hello");

List<String> stringList = rawList; // Unchecked conversion warning

System.out.println(stringList.get(0));

}

}

```

* **@FunctionalInterface:** Introduced in Java 8, this annotation is used to mark an interface as a functional interface (an interface with exactly one abstract method). The compiler will produce an error if the interface does not meet the criteria of a functional interface.

```Java

@FunctionalInterface

interface MyFunction {

int apply(int x);

}

```

**Meta-Annotations:**

Meta-annotations are annotations that are used to annotate other annotations. They provide information about how the custom annotations should be treated. The key meta-annotations are:

* **@Retention:** Specifies how long the annotation should be kept. It takes a RetentionPolicy enum as a parameter, which can be SOURCE, CLASS, or RUNTIME.
* **@Target:** Specifies the program elements to which the annotation can be applied. It takes an array of ElementType enums as a parameter (e.g., TYPE for classes, interfaces, enums; METHOD for methods; FIELD for fields; PARAMETER for parameters; etc.).
* **@Inherited:** Indicates that an annotation type can be inherited by subclasses. If a class is annotated with an @Inherited annotation, subclasses will also inherit that annotation (if the subclass is not also annotated with the same type of annotation). This applies only to class-level annotations.
* **@Documented:** Indicates that annotations of this type should be included in the generated Java documentation (Javadoc).

**Creating Custom Annotations:**

You can define your own annotation types in Java using the @interface keyword. Custom annotations can have elements that define configuration parameters.

**Syntax for Defining a Custom Annotation:**

```Java

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)

@Target({ElementType.TYPE, ElementType.METHOD})

public @interface MyCustomAnnotation {

String value() default "default value";

int count() default 0;

String[] tags() default {};

}

```

* @interface keyword is used to declare an annotation type.
* @Retention and @Target are meta-annotations that specify the retention policy and target elements for this custom annotation.
* value(), count(), and tags() are elements of the annotation. They look like methods but define the parameters that can be passed to the annotation.
* default keyword can be used to provide default values for the annotation elements.

**Using a Custom Annotation:**

```Java

@MyCustomAnnotation(value = "MyClass", count = 1, tags = {"important", "feature"})

public class MyClass {

@MyCustomAnnotation(value = "processData", count = 2)

public void processData() {

// ...

}

public static void main(String[] args) {

// You would typically use reflection to read and process custom annotations at runtime.

Class<?> clazz = MyClass.class;

MyCustomAnnotation classAnnotation = clazz.getAnnotation(MyCustomAnnotation.class);

if (classAnnotation != null) {

System.out.println("Class Annotation Value: " + classAnnotation.value());

System.out.println("Class Annotation Count: " + classAnnotation.count());

System.out.println("Class Annotation Tags: " + java.util.Arrays.toString(classAnnotation.tags()));

}

try {

java.lang.reflect.Method method = clazz.getMethod("processData");

MyCustomAnnotation methodAnnotation = method.getAnnotation(MyCustomAnnotation.class);

if (methodAnnotation != null) {

System.out.println("Method Annotation Value: " + methodAnnotation.value());

System.out.println("Method Annotation Count: " + methodAnnotation.count());

}

} catch (NoSuchMethodException e) {

e.printStackTrace();

}

}

}

```

In this example, we define a custom annotation @MyCustomAnnotation with three elements: value, count, and tags. We then apply this annotation to the MyClass and its processData method. At runtime, we use reflection to access and read the values of these annotations.

**Annotation Processing:**

While built-in annotations are often processed by the compiler or JVM, custom annotations are typically processed by annotation processing tools, either at compile time (using the Annotation Processing Tool - APT, or more recently, the Pluggable Annotation Processing API) or at runtime (using reflection).

* **Compile-time Annotation Processing:** Tools can analyze your code and annotations during compilation to generate boilerplate code, perform checks, or modify the source code. Libraries like Lombok and Dagger use compile-time annotation processing.
* **Runtime Annotation Processing:** You can use Java Reflection API to access and process annotations at runtime, as shown in the MyClass example's main method. This allows you to implement behavior based on the metadata provided by the annotations.

Annotations are a powerful mechanism for adding metadata to your Java code, enabling various forms of static analysis, code generation, and runtime behavior modification. They play a significant role in many Java frameworks and libraries.

Our next topic will be **Reflection**, which we briefly used to access annotations at runtime. Reflection is a powerful API that allows you to inspect and manipulate classes, interfaces, fields, and methods at runtime. Do you have any questions about annotations before we proceed?

**Interface**

**```Java**

abstract class Computer {

public abstract void code();

}

class Laptop extends Computer {

public void code() {

System.out.println(“code, compile, run”);

}

}

class Desktop extends Computer{

public void code() {

System.out.println(“code, compile, run, faster”);

}

}

class Developer {

public void devApp(Computer comp) {

comp.code();

}

}

class Main {

public static void main(String[] args) {

Computer lap = new Laptop();

Computer desk = new Desktop();

Developer navin = new Developer();

navin.devApp(lap);

navin.devApp(desk);

}

}

**```**

In the above example we used abstract class and method.

Instead of that we can use interface.

**```Java**

interface Computer {

public abstract void code();

}

class Laptop implements Computer {

public void code() {

System.out.println(“code, compile, run”);

}

}

class Desktop implements Computer{

public void code() {

System.out.println(“code, compile, run, faster”);

}

}

class Developer {

public void devApp(Computer comp) {

comp.code();

}

}

class Main {

public static void main(String[] args) {

Computer lap = new Laptop();

Computer desk = new Desktop();

Developer navin = new Developer();

navin.devApp(lap);

navin.devApp(desk);

}

}

**```**

**Interface is not a class**

Every method in interface are public abstract by default.

```Java

interface Computer {

public abstract void code();

public abstract void debug();

// Instead of above we can write

// even though we don’t write public abstract it is there by default

void code();

void debug;

}

public class Main {

public static void main(String[] args) {

A obj; //possible

Obj = new A(); // wrong, error.

}

}

```

We can get the reference but cannot instantiate it.

Interfaces give design support instructions not implementations

So we what we do is below:

```Java

interface Computer {

void code();

void debug;

}

class ComputerImpl implements Computer {

public void code() {

System.out.println(“Coding…”);

}

public void debug() {

System.out.println(“Debugging …”);

}

}

public class Main {

public static void main(String[] args) {

A obj;

obj = new B();

obj.code();

}

}

```

We can create variables in interface but they will be final and static.

```Java

interface Computer {

int age = 32;

String area = “Culcutta”;

void code();

void debug;

}

class ComputerImpl implements Computer {

public void code() {

System.out.println(“Coding…”);

}

public void debug() {

System.out.println(“Debugging …”);

}

}

public class Main {

public static void main(String[] args) {

A obj;

obj = new B();

obj.code();

A.area = “Mumbai”; // give you error as it is final and static

System.out.println(“A.area”); // you don’t need obj to call it

}

}

```

Interfaces don’t have that odd memory in heap as they don’t create object. We create objects of class we implements abstracts in interface.

Because we don’t instantiate the interface as object or implement the methods inside interface we need to make the variables final.

Unlike abstract we can implement multiple interfaces in one class.

```Java

interface Comp {

void code();

void debug;

}

interface CompB {

void testing();

}

class CompImpl implements Comp, CompB {

public void code() {

System.out.println(“Coding…”);

}

public void debug() {

System.out.println(“Debugging …”);

}

public void testing() {

System.out.println(“Testing …”);

}

}

```

Inheritance is valid in interfaces

```Java

interface A {

void deploy();

}

interface B extends A {

}

```

**Different types of Interfaces**

**Normal Interface:**  has multiple methods

**Functional Interface or SAM:** has only one method

**Marker Interface:** has no methods, a blank interface. Used for serialization. Stores the default values of an object in hard drive for later use.

**Functional Interface**

```Java

@FunctionalInterface

interface A {

void show();

}

public class Main {

public static void main(String[] args) {

A obj = new A() {

Public void show() {

System.out.println(“In Show Functional interface”);

}

}

}

}

``

If there is an annotation @FunctionalInterface then if we write multiple methods in that interface it will give us error before complie time.

**Lambda Expression**

These features, introduced in Java 8, have significantly changed how we write code by enabling a more concise and functional programming style.

A **lambda expression** is a concise way to represent an anonymous function (a function without a name). They are primarily used to implement methods of functional interfaces. Lambda expressions allow you to pass behavior as data, making your code more expressive and less verbose.

Used with functional interfaces other wise it will not work.

Helps syntactical sugaring.  
if you have multiple statements we need curly braces or else we can skip that.

**Examples of Lambda Expressions:**

```Java

// No parameters, returns "Hello"

() -> "Hello"

// Single parameter (type inferred), returns the parameter squared

n -> n \* n

// Two parameters with explicit types, returns their sum

(int a, int b) -> a + b

// Two parameters, block of code with a return statement

(String s1, String s2) -> {

int result = s1.compareTo(s2);

return result;

}

// Single parameter, void return type

name -> System.out.println("Hello, " + name + "!");

```

```Java

@FunctionalInterface

interface A {

void show();

}

@FunctionalInterface

interface B {

void show(int i);

}

public class Main {

public static void main(String[] args) {

A obj = () -> System.out.println(“In Show Functional interface”);

obj.show();

B obj = (int i) -> System.out.println(“In Show ” + i);

obj.show(8);

//further reduce

B obj = i -> System.out.println(“In Show ” + i);

obj.show(8);

}

}

}

```

**What if you have a return?**

If you have only one statement and that is return then you don’t have to write return

```Java

@FunctionalInterface

interface A {

int add(int I, int j);

}

public class Main {

public static void main(String[] args) {

A obj = ( I, j ) -> i+j;

Int result = obj.add(4, 5):

System.out.println(result);

}

}

}

``

A lambda expression provides the implementation for the single abstract method of a functional interface. When you assign a lambda expression to a variable of a functional interface type, you are essentially creating an instance of an anonymous class that implements that interface, and the lambda's body becomes the implementation of the abstract method.

**Example: Using Lambda Expressions with Functional Interfaces:**

```Java

public class LambdaDemo {

@FunctionalInterface

interface Calculator {

int operate(int a, int b);

}

public static void main(String[] args) {

// Implementing Calculator using a lambda expression for addition

Calculator addition = (int x, int y) -> x + y;

System.out.println("Addition: " + addition.operate(5, 3)); // Output: Addition: 8

// Implementing Calculator using a lambda expression for subtraction (type inference)

Calculator subtraction = (a, b) -> a - b;

System.out.println("Subtraction: " + subtraction.operate(10, 4)); // Output: Subtraction: 6

// Implementing Calculator using a lambda expression for multiplication (block body)

Calculator multiplication = (int a, int b) -> {

return a \* b;

};

System.out.println("Multiplication: " + multiplication.operate(2, 6)); // Output: Multiplication: 12

// Using a lambda expression with Runnable

Runnable myRunnable = () -> {

System.out.println("Running in a lambda expression!");

};

new Thread(myRunnable).start(); // Output: Running in a lambda expression! (in a new thread)

}

}

```

**Predefined Functional Interfaces in java.util.function:**

Java 8 also introduced a package java.util.function that contains many commonly used functional interfaces, so you often don't need to define your own. Some important ones include:

* **Predicate<T>:** Represents a function that accepts one argument of type T and returns a boolean result (e.g., for filtering).
* **Consumer<T>:** Represents an operation that accepts one argument of type T and performs some action, but returns no result (e.g., for printing).
* **Function<T, R>:** Represents a function that accepts one argument of type T and produces a result of type R (e.g., for transforming data).
* **Supplier<T>:** Represents a function that takes no arguments and returns a result of type T (e.g., for generating values).
* **UnaryOperator<T>:** Represents an operation on a single operand that produces a result of the same type as its operand (e.g., for incrementing).
* **BinaryOperator<T>:** Represents an operation on two operands of the same type that produces a result of the same type (e.g., for addition of two numbers).

**Method References:**

Method references are a shorthand syntax for lambda expressions that simply call an existing method. They make code even more concise and readable when you are already using a method that does exactly what you need a lambda to do.

**Types of Method References:**

1. **Static method reference:** ClassName::staticMethodName
2. **Instance method reference of a particular object:** object::instanceMethodName
3. **Instance method reference of an arbitrary object of a particular type:** ClassName::instanceMethodName
4. **Constructor reference:** ClassName::new

**Examples of Method References:**

```Java

import java.util.Arrays;

import java.util.List;

import java.util.function.Consumer;

import java.util.function.Function;

public class MethodReferenceDemo {

public static void printMessage(String msg) {

System.out.println(msg.toUpperCase());

}

public static int compareStrings(String s1, String s2) {

return s1.compareTo(s2);

}

public static void main(String[] args) {

List<String> names = Arrays.asList("alice", "bob", "charlie");

// Lambda expression to print in uppercase

names.forEach(name -> System.out.println(name.toUpperCase()));

// Method reference to a static method

names.forEach(MethodReferenceDemo::printMessage);

StringProcessor processor = new StringProcessor();

// Lambda expression using an instance method

names.forEach(name -> processor.process(name));

// Method reference to an instance method of a particular object

names.forEach(processor::process);

// Method reference to an instance method of an arbitrary object of a particular type

names.sort(String::compareTo);

System.out.println("Sorted names: " + names);

// Constructor reference

Function<String, StringBuilder> stringBuilderFactory = StringBuilder::new;

StringBuilder sb = stringBuilderFactory.apply("Hello");

System.out.println("StringBuilder: " + sb);

}

}

class StringProcessor {

public void process(String s) {

System.out.println("Processing: " + s);

}

}

```

Lambda expressions and method references are powerful features that enable functional programming paradigms in Java, leading to more concise, readable, and maintainable code, especially when working with collections and asynchronous operations.

**Exceptions**

Different types of Errors:

* Compile time erros
* Run time errors: causes the programe to stop unexpectedly. Exception handling
* Logical errors: you get an out put that you are not excepting. There will be no compile time or run time errors.

**Exception Handling**

In programming, errors can occur during the execution of a program. These errors can be due to various reasons, such as invalid user input, network issues, file not found, or logical errors in the code. Java provides a powerful mechanism called **exception handling** to deal with these runtime errors in a graceful way, preventing the program from abruptly terminating and allowing you to take corrective actions.

**What is an Exception?**

An **exception** is an event that occurs during the execution of a program that disrupts the normal flow of instructions. In Java, exceptions are objects that are instances of classes derived from the java.lang.Throwable class.

Throwable has two main subclasses:

* **Exception:** These are exceptions that a well-written program should try to catch and handle. They often indicate problems that the program can potentially recover from. Examples include IOException, SQLException, ArithmeticException, ArrayIndexOutOfBoundsException, etc.
* **Error:** These represent more serious problems that are usually beyond the control of the application and are often unrecoverable. Examples include OutOfMemoryError, StackOverflowError, VirtualMachineError, etc. Typically, you should not try to catch Errors.

**The try-catch-finally Block:**

Java provides the try-catch-finally block to handle exceptions.

```Java

try {

// Code that might throw an exception

// ...

} catch (ExceptionType1 e1) {

// Handler for ExceptionType1

// ...

} catch (ExceptionType2 e2) {

// Handler for ExceptionType2

// ...

} finally {

// Optional block that always executes, regardless of whether an exception occurred

// ...

}

```

* **try block:** This block contains the code that might potentially throw an exception. If an exception occurs within the try block, the normal flow of execution is interrupted, and the control is transferred to the corresponding catch block.
* **catch block:** This block is used to catch and handle exceptions of a specific type. You can have multiple catch blocks to handle different types of exceptions that might be thrown in the try block. The ExceptionType specifies the type of exception that this catch block can handle, and e is the reference variable that holds the exception object.
* **finally block:** This is an optional block that follows the try and any catch blocks. The code in the finally block is always executed, regardless of whether an exception occurred in the try block or was caught in a catch block. It's typically used for cleanup operations, such as closing files or releasing resources.

**Example of Exception Handling:**

```Java

public class ExceptionHandlingDemo {

public static void main(String[] args) {

int[] numbers = {1, 2, 3};

int index = 5;

try {

System.out.println("Trying to access element at index " + index);

int value = numbers[index]; // This will throw ArrayIndexOutOfBoundsException

System.out.println("Value at index " + index + ": " + value); // This line will not be reached

} catch (ArrayIndexOutOfBoundsException e) {

System.err.println("Error: Array index out of bounds!");

System.err.println("Exception details: " + e);

System.err.println("Exception message: " + e.getMessage());

e.printStackTrace(); // Prints the stack trace of the exception

} finally {

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

}

System.out.println("Program continues after exception handling.");

}

}

```

In this example:

* The try block attempts to access an element at an invalid index of the numbers array, which throws an ArrayIndexOutOfBoundsException.
* The catch block catches this specific type of exception and executes its code, printing an error message and details about the exception.
* The finally block is always executed, printing "Finally block executed."
* The program continues to execute after the try-catch-finally block.

**Checked vs. Unchecked Exceptions:**

Java distinguishes between two types of exceptions:

* **Checked Exceptions:** These are exceptions that the compiler forces you to handle (either by catching them using a try-catch block or by declaring that your method might throw them using the throws keyword). They typically represent exceptional conditions that are reasonably expected to occur and that the program should be prepared to handle. All subclasses of Exception (except RuntimeException and its subclasses) are checked exceptions (e.g., IOException, SQLException).
* **Unchecked Exceptions (Runtime Exceptions):** These are exceptions that the compiler does not force you to handle. They typically result from programming errors (bugs) and are often indicative of conditions that are unlikely to be gracefully recovered from during runtime. All subclasses of RuntimeException (e.g., ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException) and Error are unchecked.

**The throws Keyword:**

If a method might throw a checked exception that it doesn't handle itself, it must declare this using the throws keyword in its method signature. This indicates to the calling code that it needs to handle the exception.

```Java

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

public class ReadFileDemo {

public static void readFile(String filename) throws FileNotFoundException {

File file = new File(filename);

Scanner scanner = new Scanner(file); // This might throw FileNotFoundException

while (scanner.hasNextLine()) {

System.out.println(scanner.nextLine());

}

scanner.close();

}

public static void main(String[] args) {

try {

readFile("nonexistent\_file.txt");

} catch (FileNotFoundException e) {

System.err.println("Error: File not found!");

System.err.println("Details: " + e.getMessage());

}

}

}

```

In this example, the readFile method might throw a FileNotFoundException (a checked exception). Since the readFile method doesn't handle this exception, it declares it in its signature using throws FileNotFoundException. The main method then calls readFile within a try-catch block to handle the potential exception.

**Throwing Exceptions Manually:**

You can also explicitly throw exceptions in your code using the throw keyword. This is useful for signaling error conditions that your code detects.

```Java

public class VotingEligibility {

public static void checkAge(int age) {

if (age < 18) {

throw new IllegalArgumentException("Age must be 18 or older to vote.");

} else {

System.out.println("You are eligible to vote.");

}

}

public static void main(String[] args) {

try {

checkAge(15);

} catch (IllegalArgumentException e) {

System.err.println("Error: " + e.getMessage());

}

checkAge(20); // This will print "You are eligible to vote."

}

}

```

Here, if the age is less than 18, a new IllegalArgumentException (an unchecked exception) is created and thrown. The calling main method catches this exception.

**Creating Custom Exceptions:**

You can create your own exception classes by extending the Exception class (for checked exceptions) or the RuntimeException class (for unchecked exceptions). This allows you to define exceptions that are specific to your application's domain.

```Java

// Custom checked exception

class InsufficientFundsException extends Exception {

public InsufficientFundsException(String message) {

super(message);

}

}

public class BankAccount {

private double balance;

public BankAccount(double initialBalance) {

this.balance = initialBalance;

}

public void withdraw(double amount) throws InsufficientFundsException {

if (amount > balance) {

throw new InsufficientFundsException("Insufficient funds to withdraw " + amount);

}

balance -= amount;

System.out.println("Withdrawal of " + amount + " successful. New balance: " + balance);

}

public static void main(String[] args) {

BankAccount account = new BankAccount(100);

try {

account.withdraw(150);

} catch (InsufficientFundsException e) {

System.err.println("Error: " + e.getMessage());

}

try {

account.withdraw(50);

} catch (InsufficientFundsException e) {

System.err.println("Error: " + e.getMessage());

}

}

}

```

In this example, InsufficientFundsException is a custom checked exception. The withdraw method declares that it might throw this exception.

Effective exception handling is crucial for writing robust and reliable Java programs. It allows you to manage errors gracefully and prevent unexpected program termination.

**Accepting user input**

**Input/Output (I/O) Operations**

Java's I/O system is based on the concept of **streams**. A stream represents a flow of data. Java provides different types of streams to handle various kinds of data and sources/destinations.

**Types of Streams:**

Java categorizes streams into two main types:

1. **Byte Streams:** These streams are used to read and write data in the form of bytes (8-bit units). They are primarily used for binary data, such as images, audio files, and compiled code. The main abstract classes in the java.io package for byte streams are InputStream (for reading bytes) and OutputStream (for writing bytes).
   * **Common Byte Stream Classes:**
     + FileInputStream: Reads bytes from a file.
     + FileOutputStream: Writes bytes to a file.
     + ByteArrayInputStream: Reads bytes from a byte array in memory.
     + ByteArrayOutputStream: Writes bytes to a byte array in memory.
     + BufferedInputStream: Adds buffering to an input stream for efficiency.
     + BufferedOutputStream: Adds buffering to an output stream for efficiency.
     + ObjectInputStream: Reads Java objects (after deserialization).
     + ObjectOutputStream: Writes Java objects (for serialization).
2. **Character Streams:** These streams are used to read and write character data. They handle character encoding and are more suitable for text-based data. The main abstract classes in the java.io package for character streams are Reader (for reading characters) and Writer (for writing characters).
   * **Common Character Stream Classes:**
     + FileReader: Reads characters from a file.
     + FileWriter: Writes characters to a file.
     + CharArrayReader: Reads characters from a character array in memory.
     + CharArrayWriter: Writes characters to a character array in memory.
     + BufferedReader: Adds buffering to a character input stream for efficiency. Provides methods like readLine().
     + BufferedWriter: Adds buffering to a character output stream for efficiency. Provides methods like newLine().
     + InputStreamReader: Converts bytes from an input stream into characters (using a specified charset).
     + OutputStreamWriter: Converts characters to bytes in an output stream (using a specified charset).

**File I/O Operations:**

Let's focus on reading from and writing to files using both byte and character streams.

**Reading from a File (Byte Stream - FileInputStream):**

Java

import java.io.FileInputStream;

import java.io.IOException;

public class FileInputStreamDemo {

public static void main(String[] args) {

FileInputStream fis = null;

try {

fis = new FileInputStream("data.bin"); // Assuming data.bin exists

int byteRead;

while ((byteRead = fis.read()) != -1) {

System.out.print((char) byteRead); // Treat byte as character for display

}

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (fis != null) {

fis.close(); // Important to close the stream to release resources

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Writing to a File (Byte Stream - FileOutputStream):**

Java

import java.io.FileOutputStream;

import java.io.IOException;

public class FileOutputStreamDemo {

public static void main(String[] args) {

FileOutputStream fos = null;

String data = "Hello, Binary World!";

try {

fos = new FileOutputStream("output.bin");

byte[] bytes = data.getBytes(); // Convert string to bytes

fos.write(bytes);

System.out.println("Data written to output.bin");

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (fos != null) {

fos.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Reading from a File (Character Stream - FileReader and BufferedReader):**

```Java

System.out.println(“Enter a number”);

InputStreamReader in = new InputStreamReader(System.in);

BufferedReader bf = new BufferedReader(in);

int num = Integer.parseInt(bf.readLine());

System.out.println(num);

bf.close(); // best close the resource

// Best way to get user input is a scanner

Scanner sc = new Scanner(System.in);

int num = sc.nextInt();

System.out.println(num);

```

Using BufferedReader is more efficient for reading text files line by line.

```java

import java.io.BufferedReader; import java.io.FileReader; import java.io.IOException;

public class BufferedReaderDemo

{

public static void main(String[] args)

{

BufferedReader br = null;

try { br = new BufferedReader(new FileReader("text.txt")); // Assuming text.txt exists String line;

while ((line = br.readLine()) != null) {

System.out.println(line);

}

}

catch (IOException e) {

e.printStackTrace();

} finally { try { if (br != null) {

br.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

```

**Writing to a File (Character Stream - `FileWriter` and `BufferedWriter`):**

Using `BufferedWriter` allows for efficient writing and the use of `newLine()`.

```java

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

public class BufferedWriterDemo {

public static void main(String[] args) {

BufferedWriter bw = null;

String[] lines = {"First line", "Second line", "Third line"};

try {

bw = new BufferedWriter(new FileWriter("output.txt"));

for (String line : lines) {

bw.write(line);

bw.newLine(); // Adds a line separator

}

System.out.println("Data written to output.txt");

} catch (IOException e) {

e.printStackTrace();

} finally {

try {

if (bw != null) {

bw.close();

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

}

**Buffered I/O for Efficiency:**

As seen in the BufferedReader and BufferedWriter examples, using buffered streams (BufferedInputStream, BufferedOutputStream, BufferedReader, BufferedWriter) can significantly improve the performance of I/O operations, especially when dealing with large amounts of data. Buffering reduces the number of actual read/write operations to the underlying storage by reading/writing data in larger chunks.

**Try with resources**

```Java

int num = 0;

BufferedReader br = null;

try

{

br = new BufferedReader(new InputStreamReader(System.in);

num = Integer.parseInt(br.readLine());

System.out.println(num);

}

Finally

{

br.close();

}

```

```Java

int num = 0;

// New way to close the resource automatically

Try(new BufferedReader(new InputStreamReader(System.in);)

{

num = Integer.parseInt(br.readLine());

System.out.println(num);

}

```

**Serialization (Writing an Object to a File):**

```Java

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectOutputStream;

import java.io.Serializable;

class Person implements Serializable {

String name;

int age;

public Person(String name, int age) {

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "Person{name='" + name + "', age=" + age + '}';

}

}

public class SerializationDemo {

public static void main(String[] args) {

Person person = new Person("Alice", 30);

try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {

oos.writeObject(person);

System.out.println("Person object serialized to person.ser");

} catch (IOException e) {

e.printStackTrace();

}

}

}

```

**Deserialization (Reading an Object from a File):**

```Java

import java.io.FileInputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

public class DeserializationDemo {

public static void main(String[] args) {

try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream("person.ser"))) {

Person loadedPerson = (Person) ois.readObject();

System.out.println("Person object deserialized: " + loadedPerson);

} catch (IOException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

```

**Important Considerations for I/O Operations:**

* **Closing Streams:** It's crucial to close input and output streams when you are finished with them to release system resources. The finally block or try-with-resources (available since Java 7) should be used to ensure that streams are closed even if exceptions occur.
* **Exception Handling:** I/O operations are prone to IOExceptions, so proper exception handling using try-catch blocks is essential.
* **File Paths:** Be careful when specifying file paths, especially if your application needs to run on different operating systems. You might need to use platform-independent ways to construct file paths.
* **Character Encoding:** When working with character streams, be aware of character encoding (e.g., UTF-8) to ensure that text data is read and written correctly. You can specify the encoding when creating InputStreamReader and OutputStreamWriter.

Java's I/O system provides a flexible and powerful way to interact with various data sources and destinations. Understanding the difference between byte and character streams, the use of buffering, and the concepts of serialization and deserialization are important for many types of Java applications.

**Threads**

**Multithreading and Concurrency**

In traditional single-threaded programming, a program executes tasks sequentially, one after the other. **Multithreading** is a concurrency mechanism that allows multiple parts of a program to run concurrently, appearing to execute at the same time. Each concurrent part of the program is called a **thread**.

**Why Use Multithreading?**

* **Responsiveness:** For applications with graphical user interfaces (GUIs), using threads can prevent the GUI from freezing when performing long-running operations. The main thread handles the UI, while other tasks run in background threads.
* **Performance:** On multi-core processors, multithreading can significantly improve the performance of applications by allowing different threads to execute on different CPU cores simultaneously.
* **Resource Sharing:** Threads within the same process share the same memory space, which can make it easier to share data between different parts of the program.
* **Improved Throughput:** For server applications, handling each incoming request in a separate thread can increase the number of requests that can be processed concurrently, improving overall throughput.

**Threads vs. Processes:**

* **Process:** A process is an independent execution environment with its own memory space. Multiple processes can run concurrently on an operating system.
* **Thread:** A thread is a lightweight sub-process within a process. Multiple threads can exist within a single process and share the process's resources (memory, file handles, etc.).

**Creating Threads in Java:**

There are two primary ways to create threads in Java:

1. **Implementing the Runnable Interface:** You create a class that implements the java.lang.Runnable interface. This interface has a single method, run(), which contains the code that will be executed by the thread. You then create a Thread object, passing an instance of your Runnable class to its constructor, and start the thread using the start() method.

```Java

class MyRunnable implements Runnable {

private String threadName;

public MyRunnable(String name) {

threadName = name;

System.out.println("Creating " + threadName);

}

@Override

public void run() {

System.out.println("Running " + threadName);

try {

for (int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", Count: " + i);

Thread.sleep(50); // Pause the thread for 50 milliseconds

}

} catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

}

public class RunnableDemo {

public static void main(String[] args) {

MyRunnable runnable1 = new MyRunnable("Thread-1");

Thread thread1 = new Thread(runnable1);

thread1.start(); // Starts the execution of the run() method in a new thread

MyRunnable runnable2 = new MyRunnable("Thread-2");

Thread thread2 = new Thread(runnable2);

thread2.start();

}

}

```

**Lambda version of Runnable**

```Java

Runnable obj 1 = () -> {

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

System.out.println(“Hi”);

Try {Thread.sleep(10); }

catch (InterruptedException e) {e.message; }

}

}

```

1. **Extending the Thread Class:** You create a class that extends the java.lang.Thread class and override its run() method. You then create an instance of your subclass and call its start() method.

```Java

class MyThread extends Thread {

private String threadName;

public MyThread(String name) {

threadName = name;

System.out.println("Creating " + threadName);

}

@Override

public void run() {

System.out.println("Running " + threadName);

try {

for (int i = 4; i > 0; i--) {

System.out.println("Thread: " + threadName + ", Count: " + i);

Thread.sleep(100);

}

} catch (InterruptedException e) {

System.out.println("Thread " + threadName + " interrupted.");

}

System.out.println("Thread " + threadName + " exiting.");

}

}

public class ThreadDemo {

public static void main(String[] args) {

MyThread thread1 = new MyThread("Thread-A");

thread1.start();

MyThread thread2 = new MyThread("Thread-B");

thread2.start();

}

}

```

**Choosing between Runnable and Thread:**

Implementing Runnable is generally preferred because:

* A class can implement multiple interfaces, but it can extend only one class. If your class already extends another class, you must use Runnable.
* It promotes better separation of concerns: your task logic is in the Runnable class, and the thread management is done by the Thread object.

**Thread Synchronization:**

When multiple threads access shared resources (like variables or data structures), it can lead to **race conditions** where the outcome of the program depends on the unpredictable order in which the threads execute. To prevent this, you need to **synchronize** access to shared resources.

Java provides several mechanisms for thread synchronization:

* **synchronized Keyword:** You can use the synchronized keyword to create synchronized methods or synchronized blocks. Only one thread can execute a synchronized method or block on a given object at a time. The synchronized keyword acquires a lock on the object (or the class object for static synchronized methods).

```Java

class Counter {

private int count = 0;

// Synchronized method

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

public class SynchronizationDemo {

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

Counter counter = new Counter();

Runnable task = () -> {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

};

Thread thread1 = new Thread(task);

Thread thread2 = new Thread(task);

thread1.start();

thread2.start();

thread1.join(); // Wait for thread1 to finish

thread2.join(); // Wait for thread2 to finish

System.out.println("Final count: " + counter.getCount()); // Should be 2000

}

}

```

* **wait(), notify(), and notifyAll() Methods:** These methods are provided by the Object class and are used for inter-thread communication within synchronized blocks or methods.
  + wait(): Causes the current thread to release the lock on the object and wait until another thread notifies it.
  + notify(): Wakes up a single thread that is waiting on the object's monitor.
  + notifyAll(): Wakes up all threads that are waiting on the object's monitor.

```Java

class Message {

private String msg;

private boolean empty = true;

public synchronized void put(String newMsg) {

while (!empty) {

try {

wait(); // Wait until the message is consumed

} catch (InterruptedException e) {

// ...

}

}

this.msg = newMsg;

empty = false;

notifyAll(); // Notify waiting consumer

}

public synchronized String take() {

while (empty) {

try {

wait(); // Wait until a message is available

} catch (InterruptedException e) {

// ...

}

}

empty = true;

notifyAll(); // Notify waiting producer

return this.msg;

}

}

class Producer implements Runnable {

private Message msg;

public Producer(Message m) {

this.msg = m;

}

@Override

public void run() {

String[] messages = {"Message 1", "Message 2", "Message 3", "DONE"};

for (String message : messages) {

System.out.println("Producer putting: " + message);

msg.put(message);

try {

Thread.sleep(100);

} catch (InterruptedException e) {

// ...

}

}

}

}

class Consumer implements Runnable {

private Message msg;

public Consumer(Message m) {

this.msg = m;

}

@Override

public void run() {

String message;

while (!(message = msg.take()).equals("DONE")) {

System.out.println("Consumer got: " + message);

try {

Thread.sleep(50);

} catch (InterruptedException e) {

// ...

}

}

}

}

public class WaitNotifyDemo {

public static void main(String[] args) {

Message msg = new Message();

new Thread(new Producer(msg)).start();

new Thread(new Consumer(msg)).start();

}

}

```

**Understanding and Avoiding Deadlocks:**

A **deadlock** occurs when two or more threads are blocked indefinitely, waiting for each other to release the resources that they need. Deadlocks can be a serious problem in concurrent applications. To avoid deadlocks, you need to ensure that threads acquire locks in a consistent order and release them properly.

**The java.util.concurrent Package:**

Java 5 introduced the java.util.concurrent package, which provides a rich set of high-level concurrency utilities that are often easier and safer to use than the basic thread creation and synchronization mechanisms. Some key components include:

* **Executors:** An interface for managing and controlling thread execution. The Executors class provides factory methods to create various thread pool implementations (e.g., newFixedThreadPool(), newCachedThreadPool(), newSingleThreadExecutor()).

```Java

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ExecutorDemo {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(2); // Create a thread pool with 2 threads

Runnable task1 = () -> {

System.out.println("Task 1 running in " + Thread.currentThread().getName());

try {

Thread.sleep(200);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

};

Runnable task2 = () -> {

System.out.println("Task 2 running in " + Thread.currentThread().getName());

try {

Thread.sleep(300);

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

};

executor.submit(task1); // Submit tasks to the executor

executor.submit(task2);

executor.shutdown(); // Signal that no new tasks will be submitted

try {

executor.awaitTermination(1, java.util.concurrent.TimeUnit.SECONDS); // Wait for tasks to complete

} catch (InterruptedException e) {

Thread.currentThread().interrupt();

}

}

}

```

* **Future:** Represents the result of an asynchronous computation. You can submit a Callable (a Runnable that can return a result and throw exceptions) to an ExecutorService and get a Future object. You can then use the Future to check if the computation is complete, retrieve the result (which might block until it's available), or cancel the task.

```Java

import java.util.concurrent.Callable;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.Future;

public class FutureDemo {

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

ExecutorService executor = Executors.newSingleThreadExecutor();

Callable<Integer> task = () -> {

System.out.println("Calculating sum in " + Thread.currentThread().getName());

Thread.sleep(1000);

return 10 + 20;

};

Future<Integer> future = executor.submit(task);

System.out.println("Task submitted. Waiting for result...");

Integer result = future.get(); // Blocks until the result is available

System.out.println("Result is: " + result);

executor.shutdown();

}

}

```

* **Synchronization Utilities:** The java.util.concurrent.locks package provides more flexible locking mechanisms (like ReentrantLock) and condition variables (Condition). The java.util.concurrent.atomic package provides atomic variables that support thread-safe operations without explicit locking. The java.util.concurrent.CountDownLatch, CyclicBarrier, Semaphore, and Exchanger classes provide utilities for coordinating the execution of multiple threads.

Multithreading and concurrency are powerful tools for building efficient and responsive applications, but they also introduce complexities related to shared resources, synchronization, and potential deadlocks. Understanding these concepts and using the appropriate tools from the java.util.concurrent package is crucial for writing correct and scalable concurrent programs.

**Collections Framework**

**Working with Collections**

The **Java Collections Framework** is a powerful and essential part of the Java language. It provides a set of **interfaces** and **classes** for representing and manipulating collections of objects. Think of collections as containers that can hold multiple items, similar to how arrays work, but with more flexibility and functionality.

**Why do we need Collections?**

* **Dynamic Size:** Unlike arrays, many collections can grow or shrink in size dynamically as needed.
* **Rich Functionality:** The Collections Framework provides various data structures with built-in methods for common operations like adding, removing, searching, sorting, and iterating over elements.
* **Organization and Management:** Collections help organize and manage large amounts of data in a structured and efficient way.
* **Reusability:** The framework provides well-tested and optimized implementations of common data structures, promoting code reuse.

**Core Interfaces of the Collections Framework:**

The Collections Framework is built around a set of core interfaces that define the basic operations for different types of collections. Some of the most important interfaces are:

* **Collection:** The root interface in the hierarchy. It represents a group of objects (the "elements" of the collection). It defines basic operations that all collections should support, such as add(), remove(), contains(), size(), isEmpty(), and iterator().
* **List:** An ordered collection (also known as a sequence). Elements can be accessed by their integer index (position in the list). Lists can contain duplicate elements. Key implementations include ArrayList, LinkedList, and Vector.
* **Set:** A collection that contains no duplicate elements. Sets do not guarantee any particular order of elements (unless specific implementations like LinkedHashSet or TreeSet are used). Key implementations include HashSet, LinkedHashSet, and TreeSet.
* **Map:** An object that maps keys to values. Each key can map to at most one value. Maps do not guarantee any particular order of key-value pairs (unless specific implementations like LinkedHashMap or TreeMap are used). Key implementations include HashMap, LinkedHashMap, and TreeMap. Note that Map does not extend the Collection interface; it's a separate interface in the framework.

**Common Implementations:**

**Collection:**

Is an interface.

Collection extends Iterable.

If we don’t provide the data type in the < > brackets as generics it will give us objects.

Collection does not support index value method.

If you just want to add values and fetch use collection.

```Java

Collection nums = new ArrayList();

nums.add(5);

nums.add(3);

nums.add(8);

nums.add(2);

for(Objects n: nums) {

int num = (Integer)n;

System.out.println(num);

}

Collection<Integer> nums = new ArrayList<Integer>();

nums.add(5);

nums.add(3);

nums.add(8);

nums.add(2);

for(Int n: nums) {

System.out.println(n);

}

```

ArrayList is a class which implements List.

**List**

List extends Collection.

If you want to deal with index values use List.

**List Implementations:**

* **ArrayList:** A resizable array implementation of the List interface. It provides fast random access to elements (by index) but can be slower for insertions and deletions in the middle of the list because elements might need to be shifted.

```Java

import java.util.ArrayList;

import java.util.List;

public class ArrayListDemo {

public static void main(String[] args) {

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

names.add("Alice");

names.add("Bob");

names.add("Charlie");

names.add("Bob"); // Duplicates are allowed

System.out.println(names); // Output: [Alice, Bob, Charlie, Bob]

System.out.println("Size: " + names.size()); // Output: Size: 4

System.out.println("Element at index 1: " + names.get(1)); // Output: Element at index 1: Bob

names.remove("Bob"); // Removes the first occurrence of "Bob"

System.out.println(names); // Output: [Alice, Charlie, Bob]

names.add(1, "David"); // Inserts "David" at index 1

System.out.println(names); // Output: [Alice, David, Charlie, Bob]

}

}

```

**List with Linked List**

**LinkedList**: A doubly-linked list implementation of the List interface. It provides efficient insertions and deletions at any position in the list but can be slower for random access compared to ArrayList because you might need to traverse the list to reach a specific index.

```Java

import java.util.LinkedList; import java.util.List;

public class LinkedListDemo {

public static void main(String[] args) {

List<String> tasks = new LinkedList<>();

tasks.add("Task 1");

tasks.add("Task 2");

tasks.addFirst("Task 0"); // Adds to the beginning

tasks.addLast("Task 3"); // Adds to the end

System.out.println(tasks); // Output: [Task 0, Task 1, Task 2, Task 3]

System.out.println("First task: " + tasks.getFirst()); // Output: First task: Task 0

tasks.remove(1); // Removes the element at index 1

System.out.println(tasks); // Output: [Task 0, Task 2, Task 3]

}

}

```

**Vector**

Similar to ArrayList, but Vector is synchronized (thread-safe), which means it can be safely used in multithreaded environments. However, this synchronization comes with a performance overhead, so ArrayList is generally preferred in single-threaded scenarios. Vector is one of the legacy collection classes.

**Set Implementations:**

Set- unique values.

Set extends Collection.

Set will not give you values in sorted format.

Set does not support index values.

* **HashSet:** An implementation of the Set interface that uses a hash table for storing elements. It provides very fast (constant time on average) performance for adding, removing, and checking if an element exists. HashSet does not guarantee any order of elements.

```java

import java.util.HashSet; import java.util.Set;

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

{

Set<String> uniqueNames = new HashSet<>();

uniqueNames.add("Alice");

uniqueNames.add("Bob"); uniqueNames.add("Charlie"); uniqueNames.add("Alice"); // Duplicate, will not be added

System.out.println(uniqueNames); // Output: [Alice, Charlie, Bob] (order may vary)

System.out.println("Contains 'Bob': " + uniqueNames.contains("Bob")); // Output: Contains 'Bob': true

}

}

```

* **LinkedHashSet:** An implementation of the Set interface that maintains the insertion order of elements. It uses a hash table for efficient lookups and a linked list to maintain the order.

```Java

import java.util.LinkedHashSet; import java.util.Set;

public class LinkedHashSetDemo {

public static void main(String[] args) {

Set<String> orderedNames = new LinkedHashSet<>();

orderedNames.add("Alice");

orderedNames.add("Bob");

orderedNames.add("Charlie");

orderedNames.add("Alice"); // Duplicate, not added

System.out.println(orderedNames); // Output: [Alice, Bob, Charlie] (insertion order preserved)

}

}

```

**TreeSet:** An implementation of the Set interface that stores elements in a sorted order (either natural ordering if the elements implement Comparable, or using a Comparator provided at the time of creation). It uses a tree data structure (specifically, a red-black tree).

Tree set extends Abstract set which extend Navigational set and sorted set.

We can use Collection with TreeSet. Which helps us to use Iterator.

```Java

import java.util.Set;

import java.util.TreeSet;

public class TreeSetDemo {

public static void main(String[] args) {

Set<Integer> sortedNumbers = new TreeSet<>();

sortedNumbers.add(5);

sortedNumbers.add(1);

sortedNumbers.add(3);

sortedNumbers.add(5); // Duplicate, not added

System.out.println(sortedNumbers); // Output: [1, 3, 5] (sorted order)

}

}

```

**Map Implementations:**

**Map** is an interface.

As it works with key value pairs the key does not support duplicates. Values can be updated or repeated.

Its, key-value, more like a combination of set and list.

* **HashMap:** An implementation of the Map interface that uses a hash table to store key-value pairs. It provides very fast (constant time on average) performance for basic operations like get(), put(), and remove(). HashMap does not guarantee any order of entries.

```Java

import java.util.HashMap; import java.util.Map;

public class HashMapDemo {

public static void main(String[] args) {

Map<String, Integer> ages = new HashMap<>();

ages.put("Alice", 30);

ages.put("Bob", 25);

ages.put("Charlie", 35);

ages.put("Alice", 31); // Value for "Alice" is updated

System.out.println(ages); // Output: {Alice=31, Charlie=35, Bob=25} (order may vary)

System.out.println("Age of Bob: " + ages.get("Bob")); // Output: Age of Bob: 25

System.out.println("Contains key 'Alice': " + ages.containsKey("Alice")); // Output: true

for(String key: ages.keySet()) {

System.out.println(key + “:” + ages.get(key)

}

}

}

```

**LinkedHashMap:** An implementation of the Map interface that maintains the insertion order of entries (or the order in which keys were last accessed, if specified). It uses a hash table for fast lookups and a linked list to maintain the order.

```java

import java.util.LinkedHashMap;

import java.util.Map;

public class LinkedHashMapDemo {

public static void main(String[] args) {

Map<String, Integer> orderedAges = new LinkedHashMap<>(); orderedAges.put("Alice", 30);

orderedAges.put("Bob", 25);

orderedAges.put("Charlie", 35);

System.out.println(orderedAges); // Output: {Alice=30, Bob=25, Charlie=35} (insertion order)

}

}

```

* **TreeMap:** An implementation of the Map interface that stores entries in a sorted order based on the keys (either natural ordering if keys implement Comparable, or using a Comparator). It uses a tree data structure.

```Java

import java.util.Map; import java.util.TreeMap;

public class TreeMapDemo {

public static void main(String[] args) {

Map<String, Integer> sortedAges = new TreeMap<>();

sortedAges.put("Charlie", 35);

sortedAges.put("Alice", 30);

sortedAges.put("Bob", 25);

System.out.println(sortedAges); // Output: {Alice=30, Bob=25, Charlie=35} (sorted by key)

}

}

```

**Iterating Through Collections:**

There are several ways to iterate over elements in a collection:

* **Using an Iterator:** The Iterator interface provides a way to traverse the elements of a collection.

```Java

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

colors.add("Red");

colors.add("Green");

colors.add("Blue");

java.util.Iterator<String> iterator = colors.iterator();

while (iterator.hasNext()) {

String color = iterator.next();

System.out.println(color);

if (color.equals("Green")) {

iterator.remove(); // Safely remove an element during iteration

}

}

System.out.println(colors); // Output: [Red, Blue]

```

* **Using the Enhanced for loop (for-each loop):** This is a more concise way to iterate over collections that implement the Iterable interface (which all standard collections do).

```Java

List<Integer> numbers = new ArrayList<>();

numbers.add(10);

numbers.add(20);

numbers.add(30);

for (int number : numbers) {

System.out.println(number);

}

```

* **Using forEach() method (Java 8+):** Collections provide a forEach() method that accepts a lambda expression or a method reference to perform an action on each element.

```Java

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.forEach(fruit -> System.out.println(fruit));

// Or using a method reference:

fruits.forEach(System.out::println);

```

**Sorting**

We have a class Collections which helps us sort a list.

For custom logic while sorting use comparator.

As Comparator is an interface we can use anonymous class implementation.

Here you have to mention identifier type as well.

```Java

Comparator<Integer> comp = new Comparator<Integer> {

public int compare(Integer i, Integer j) {

if(i%10 > j%10)

return 1;

else

return -1;

}

}

List<Integer> nums = new ArrayList<>();

fruits.add(12);

fruits.add(10);

fruits.add(15);

Collections.sort(num, comp);

```

```Java

Class Student {

int age;

String name;

public Student (int age, String name) {

this.age = age;

this.name = name;

}

public String toString() {

return “Student [age= ” + age + “, name=” + name +”]”

}

}

// Main class

Comparator<Student> comp = new Comparator<Student> {

public int compare(Student i, Student j) {

if(i.age > j.age)

return 1;

else

return -1;

}

}

// Lambda – because Comparator is a Functional interface

Comparator<Student> comp = (i, j) -> i.age > j.age ? 1:-1;

List<Student> studs = new ArrayList<>();

studs.add(new Student(22, “Manu”));

studs.add(new Student(25, “Ben”));

studs.add(new Student(24, “John”));

studs.add(new Student(23, “Job”));

for (Student s : studs) {

System.out.println(s);

}

Collections.sort(studs, comp);

```

**Generics**

**Generics** were introduced in Java 5 to provide **compile-time type safety**. They allow you to parameterize classes, interfaces, and methods with type parameters. This means that you can define a class or a method that can work with different types of objects while ensuring type correctness at compile time.

**Generics in Collections:**

Notice the use of angle brackets <> when declaring collections (e.g., List<String>, Set<Integer>, Map<String, Integer>). This is called **generics**. Generics provide type safety by allowing you to specify the type of objects that a collection can hold. This helps in catching type errors at compile time rather than runtime and makes your code more readable and maintainable.

We will delve deeper into generics later, but it's important to use them with collections to ensure type safety.

The Java Collections Framework is a vast and powerful tool. Understanding the core interfaces and their common implementations is crucial for efficient data management in your Java applications.

**Why Use Generics?**

* **Type Safety:** Generics enable the compiler to detect type mismatches at compile time, reducing the risk of ClassCastException errors at runtime.
* **Code Reusability:** You can write a single class or method that can work with various types of objects without needing to write specific versions for each type.
* **Readability:** Generics make code easier to read and understand because the type of objects being used is explicitly specified.

**Generic Classes:**

A generic class is a class that is defined with type parameters. When you create an instance of a generic class, you provide the actual type(s) to be used for these type parameters.

**Syntax for a Generic Class:**

```Java

class ClassName<T> {

private T data;

public ClassName(T data) {

this.data = data;

}

public T getData() {

return data;

}

public void setData(T data) {

this.data = data;

}

}

```

* <T> is a type parameter. You can use one or more type parameters, separated by commas (e.g., <T, U, V>). By convention, type parameters are single uppercase letters. Common ones include T (Type), E (Element), K (Key), V (Value), N (Number).

**Using a Generic Class:**

```Java

public class GenericClassDemo {

public static void main(String[] args) {

// Create an instance of GenericClass that holds an Integer

ClassName<Integer> integerBox = new ClassName<>(10);

int value = integerBox.getData(); // No need for casting

System.out.println("Integer value: " + value);

// Create an instance of GenericClass that holds a String

ClassName<String> stringBox = new ClassName<>("Hello");

String text = stringBox.getData(); // No need for casting

System.out.println("String value: " + text);

// You cannot put a String into an Integer box (compile-time error)

// integerBox.setData("World");

}

}

```

In this example, ClassName<T> is a generic class. When we create integerBox, we specify Integer as the type parameter, so T is replaced with Integer. Similarly, for stringBox, T is replaced with String. This ensures type safety: you can't accidentally put a String into an Integer box because the compiler will catch the error.

**Generic Interfaces:**

You can also define generic interfaces.

**Syntax for a Generic Interface:**

```Java

interface MyInterface<T> {

void process(T value);

T getResult();

}

```

**Implementing a Generic Interface:**

```Java

class IntegerProcessor implements MyInterface<Integer> {

private Integer result;

@Override

public void process(Integer value) {

result = value \* 2;

}

@Override

public Integer getResult() {

return result;

}

}

class StringReverser implements MyInterface<String> {

private String result;

@Override

public void process(String value) {

result = new StringBuilder(value).reverse().toString();

}

@Override

public String getResult() {

return result;

}

}

public class GenericInterfaceDemo {

public static void main(String[] args) {

IntegerProcessor intProcessor = new IntegerProcessor();

intProcessor.process(5);

System.out.println("Processed integer: " + intProcessor.getResult()); // Output: 10

StringReverser strReverser = new StringReverser();

strReverser.process("Java");

System.out.println("Reversed string: " + strReverser.getResult()); // Output: avaJ

}

}

```

**Generic Methods:**

You can define methods that are generic, even if the class they belong to is not generic. The type parameters for a generic method are declared in the method signature.

**Syntax for a Generic Method:**

```Java

<T> returnType methodName(T parameter) {

// ...

}

<T, U> returnType anotherMethod(T param1, U param2) {

// ...

}

```

**Using a Generic Method:**

```Java

public class GenericMethodDemo {

// A generic method to print the elements of an array of any type

public static <E> void printArray(E[] array) {

for (E element : array) {

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

}

System.out.println();

}

public static void main(String[] args) {

Integer[] intArray = {1, 2, 3, 4, 5};

String[] stringArray = {"Hello", "World"};

printArray(intArray); // Output: 1 2 3 4 5

printArray(stringArray); // Output: Hello World

}

}

```

The type parameter <E> is declared before the return type void. The compiler infers the actual type of E based on the argument passed to the method.

**Bounded Type Parameters:**

You can restrict the types that can be used as type arguments by using bounded type parameters with the extends keyword. This allows you to work with types that are subtypes of a specific class or implement a specific interface.

**Example of Bounded Type Parameters:**

```Java

class NumberBox<T extends Number> {

private T number;

public NumberBox(T number) {

this.number = number;

}

public double getDoubleValue() {

return number.doubleValue(); // We can call methods of Number

}

}

public class BoundedTypeDemo {

public static void main(String[] args) {

NumberBox<Integer> intBox = new NumberBox<>(10);

System.out.println("Integer as double: " + intBox.getDoubleValue()); // Output: 10.0

NumberBox<Double> doubleBox = new NumberBox<>(3.14);

System.out.println("Double value: " + doubleBox.getDoubleValue()); // Output: 3.14

// NumberBox<String> stringBox = new NumberBox<>("Hello"); // Compile-time error: String is not a subtype of Number

}

}

```

Here, <T extends Number> means that T must be a subtype of the Number class (e.g., Integer, Double, Float). This allows us to call methods defined in the Number class on the number field.

You can also use extends with interfaces: <T extends Comparable<T>> means T must implement the Comparable interface. If you need to specify multiple bounds (e.g., a type that extends a class and implements an interface), you use &: <T extends ClassA & InterfaceB>.

**Wildcards:**

Wildcards (?) are used as type arguments when you don't know the exact type or when you want to specify a range of types.

* **Unbounded Wildcard (<?>):** Represents an unknown type. It can be useful when you want to write code that works with any type.

```Java

public static void printList(List<?> list) { for (Object item : list) { System.out.print(item + " "); } System.out.println(); }

```

* **Upper-Bounded Wildcard (<? extends SomeClass>):** Represents an unknown type that is a subtype of SomeClass.

```Java

public static double sumOfList(List<? extends Number> list) {

double sum = 0;

for (Number n : list) {

sum += n.doubleValue();

}

return sum;

}

```

* **Lower-Bounded Wildcard (<? super SomeClass>):** Represents an unknown type that is a supertype of SomeClass. This is less common but can be useful when you want to write data to a collection of an unknown supertype.

```Java

public static void addIntegers(List<? super Integer> list) {

list.add(1);

list.add(2);

}

```

**Type Erasure:**

It's important to note that Java implements generics using a mechanism called **type erasure**. At runtime, the type parameters are erased, and all generic types are treated as their raw types (e.g., List<String> becomes just List). The type checking is primarily done at compile time. Type erasure ensures backward compatibility with older versions of Java that did not have generics. However, it also means that some type-specific operations are not possible at runtime.

Generics are a fundamental feature in modern Java programming, providing significant benefits in terms of type safety, code reusability, and readability. Understanding how to use generic classes, interfaces, and methods, as well as bounded types and wildcards, is crucial for writing robust and maintainable Java code.

**Stream API**

**Java Streams API**

The **Java Streams API** provides an abstraction for processing a sequence of elements. A stream is not a data structure that stores elements; instead, it conveys elements from a source (like a collection, array, or I/O resource) through a pipeline of operations to produce a result.

**Key Characteristics of Streams:**

* **Sequence of elements:** A stream provides a sequence of elements of a specific type.
* **Source:** Streams consume elements from a source, such as a collection, array, or I/O resource.
* **Pipeline:** Stream operations are chained together to form a pipeline that processes the elements.
* **Data processing operations:** Streams support various operations like filtering, mapping, sorting, reducing, etc.
* **Lazy evaluation:** Many stream operations are lazy, meaning they are only executed when a terminal operation is invoked. This allows for efficient processing of large datasets.
* **No data modification:** Stream operations typically do not modify the original data source. Instead, they return new streams or results.
* **Consumption:** Streams can only be consumed once. After a terminal operation is performed, the stream is closed, and you need to create a new stream from the source if you want to process the same data again.

**Stream Operations:**

Stream operations are divided into two types:

1. **Intermediate Operations:** These operations transform or filter the stream and return a new stream. Multiple intermediate operations can be chained together. Examples include filter(), map(), flatMap(), sorted(), distinct(), peek(), limit(), skip().
2. **Terminal Operations:** These operations produce a result or a side-effect. Once a terminal operation is invoked, the stream is consumed and cannot be used again. Examples include forEach(), toArray(), collect(), reduce(), count(), min(), max(), findFirst(), findAny(), anyMatch(), allMatch(), noneMatch().

**Creating Streams:**

You can create streams from various sources:

* **From Collections:** Use the stream() method of a Collection.

```Java

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

Stream<String> nameStream = names.stream();

```

* **From Arrays:** Use Arrays.stream() or Stream.of().

```Java

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

IntStream numberStream = Arrays.stream(numbers);

Stream<Integer> numberStream2 = Stream.of(1, 2, 3, 4, 5);

```

* **From Stream.of():** To create a stream of individual elements.

```Java

Stream<String> messageStream = Stream.of("Hello", "World");

```

* **From Stream.generate():** To create an infinite stream using a Supplier.

```Java

Stream<Double> randomStream = Stream.generate(Math::random).limit(5);

// Limit to 5 elements

```

* **From Stream.iterate():** To create an infinite stream by iteratively applying a function.

```Java

Stream<Integer> evenNumbers = Stream.iterate(0, n -> n + 2).limit(5); // 0, 2, 4, 6, 8

* **From I/O resources:** For example, reading lines from a file using Files.lines().

try (Stream<String> lines = Files.lines(Paths.get("file.txt"))) { lines.forEach(System.out::println); } catch (IOException e) { e.printStackTrace(); }

```

**Common Stream Operations with Examples:**

Let's see some examples of intermediate and terminal operations:

**1. filter():** Filters elements based on a Predicate.

```Java

List<String> words = Arrays.asList("apple", "banana", "apricot", "kiwi");

words.stream()

.filter(s -> s.startsWith("a"))

.forEach(System.out::println); // Output: apple, apricot

**```**

**2. map():** Transforms each element using a Function.

```Java

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

names.stream()

.map(String::toUpperCase)

.forEach(System.out::println); // Output: ALICE, BOB, CHARLIE

```

**3. flatMap():** Transforms each element to a stream and then flattens the resulting streams into one. Useful for working with collections of collections.

```Java

List<List<Integer>> listOfLists = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4, 5), Arrays.asList(6));

listOfLists.stream()

.flatMap(List::stream)

.forEach(System.out::println); // Output: 1, 2, 3, 4, 5, 6

```

**4. sorted():** Sorts the elements of the stream. Can take a Comparator as an argument.

```Java

List<Integer> numbers = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6);

numbers.stream()

.sorted()

.forEach(System.out::println); // Output: 1, 1, 2, 3, 4, 5, 6, 9

List<String> words = Arrays.asList("apple", "banana", "kiwi");

words.stream()

.sorted((s1, s2) -> s2.compareTo(s1)) // Sort in reverse order

.forEach(System.out::println); // Output: kiwi, banana, apple

```

**5. distinct():** Returns a stream with duplicate elements removed (based on equals() method).

```Java

List<Integer> numbersWithDuplicates = Arrays.asList(1, 2, 2, 3, 1, 4);

numbersWithDuplicates.stream()

.distinct()

.forEach(System.out::println); // Output: 1, 2, 3, 4

**```**

**6. limit(n):** Returns a stream containing the first n elements.

```Java

Stream.iterate(1, n -> n + 1)

.limit(5)

.forEach(System.out::println); // Output: 1, 2, 3, 4, 5

**```**

**7. skip(n):** Skips the first n elements and returns the rest of the stream.

```Java

List<String> colors = Arrays.asList("red", "green", "blue", "yellow");

colors.stream()

.skip(2)

.forEach(System.out::println); // Output: blue, yellow

```

**8. forEach(action):** Performs an action for each element of the stream (terminal operation).

```Java

List<String> greetings = Arrays.asList("Hello", "Bonjour", "Hola");

greetings.stream()

.forEach(System.out::println);

```

**9. toArray():** Returns an array containing the elements of the stream (terminal operation).

```Java

List<Integer> nums = Arrays.asList(10, 20, 30);

Integer[] numsArray = nums.stream().toArray(Integer[]::new);

System.out.println(Arrays.toString(numsArray)); // Output: [10, 20, 30]

**```**

**10. collect(collector):** Accumulates elements into a result container (terminal operation). The Collectors class provides many useful collectors.

```Java

List<String> fruits = Arrays.asList("apple", "banana", "orange");

List<String> upperCaseFruits = fruits.stream()

.map(String::toUpperCase)

.collect(Collectors.toList());

System.out.println(upperCaseFruits); // Output: [APPLE, BANANA, ORANGE]

Set<String> fruitSet = fruits.stream()

.collect(Collectors.toSet());

System.out.println(fruitSet); // Output: [banana, orange, apple] (order may vary)

String joinedFruits = fruits.stream()

.collect(Collectors.joining(", "));

System.out.println(joinedFruits); // Output: apple, banana, orange

Map<String, Integer> fruitLengths = fruits.stream()

.collect(Collectors.toMap(Function.identity(), String::length));

System.out.println(fruitLengths); // Output: {apple=5, banana=6, orange=6}

long count = fruits.stream()

.filter(s -> s.startsWith("a"))

.count();

System.out.println("Number of fruits starting with 'a': " + count); // Output: 1

```

**11. reduce(identity, accumulator):** Performs a reduction on the elements of the stream using an associative accumulation function (terminal operation).

```Java

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

int sum = numbers.stream()

.reduce(0, (a, b) -> a + b); // Identity is 0, accumulator adds

System.out.println("Sum: " + sum); // Output: 15

Optional<Integer> product = numbers.stream()

.reduce((a, b) -> a \* b); // No identity, returns an Optional

product.ifPresent(p -> System.out.println("Product: " + p)); // Output: Product: 120

```

**12. min(comparator), max(comparator):** Returns the minimum or maximum element of the stream according to the provided Comparator (terminal operation). Returns an Optional.

```Java

List<Integer> nums = Arrays.asList(3, 1, 4, 1, 5, 9, 2, 6);

Optional<Integer> min = nums.stream().min(Integer::compareTo);

min.ifPresent(m -> System.out.println("Min: " + m)); // Output: Min: 1

Optional<Integer> max = nums.stream().max(Integer::compareTo);

max.ifPresent(m -> System.out.println("Max: " + m)); // Output: Max: 9

```

**13. findFirst(), findAny():** Returns an Optional describing the first or any element of the stream (terminal operation). findAny() might be more efficient in parallel streams.

```Java

Optional<String> firstFruit = fruits.stream().findFirst();

firstFruit.ifPresent(f -> System.out.println("First fruit: " + f)); // Output: First fruit: apple

Optional<String> anyFruitStartingWithB = fruits.stream().filter(s -> s.startsWith("b")).findAny();

anyFruitStartingWithB.ifPresent(f -> System.out.println("Any fruit starting with 'b': " + f)); // Output: banana

```

**14. anyMatch(predicate), allMatch(predicate), noneMatch(predicate):** Check if any, all, or none of the elements in the stream match the provided Predicate (terminal operations).

```Java

boolean anyStartsWithA = fruits.stream().anyMatch(s -> s.startsWith("a")); // true

boolean allUpperCase = fruits.stream().allMatch(s -> s.equals(s.toUpperCase())); // false

boolean noneEndsWithZ = fruits.stream().noneMatch(s -> s.endsWith("z")); // true

```

**Parallel Streams:**

The Streams API also supports parallel processing of streams, which can significantly improve performance for large datasets on multi-core processors. You can obtain a parallel stream from a collection by calling parallelStream() instead of stream(), or by calling the parallel() method on a sequential stream.

```Java

List<Integer> numbers = IntStream.rangeClosed(1, 1000000).boxed().collect(Collectors.toList());

long startTime = System.currentTimeMillis();

long sumSequential = numbers.stream().reduce(0, Integer::sum);

long endTimeSequential = System.currentTimeMillis();

System.out.println("Sequential sum: " + sumSequential + ", time: " + (endTimeSequential - startTime) + " ms");

startTime = System.currentTimeMillis();

long sumParallel = numbers.parallelStream().reduce(0, Integer::sum);

long endTimeParallel = System.currentTimeMillis();

System.out.println("Parallel sum: " + sumParallel + ", time: " + (endTimeParallel - startTime) + " ms");

```

**Primitive Streams:**

Java also provides primitive stream specializations for int, long, and double: IntStream, LongStream, and DoubleStream. These streams offer specialized operations for primitive types and can be more efficient when dealing with large sequences of numbers. You can convert between object streams and primitive streams using methods like mapToInt(), mapToLong(), mapToDouble(), boxed().

The Java Streams API is a powerful tool for processing data in a functional and concise way. It encourages a declarative style of programming, where you specify what you want to achieve rather than how to achieve it. Understanding and utilizing streams can lead to more readable, maintainable, and efficient code, especially when dealing with collections of data.

**Reflection**

As we briefly saw with annotations, reflection is a powerful API that allows you to inspect and manipulate classes, interfaces, fields, and methods at runtime.

**What is Reflection?**

Reflection is a feature in Java that allows a running Java program to examine and modify the behavior of classes, interfaces, fields, and methods at runtime. Without knowing the names of the classes, methods, fields, etc., at compile time, you can discover and interact with them during program execution.

Think of it as having the ability to look inside a class and see its structure (fields, methods, constructors), and even invoke its methods or modify its fields, all while the program is running.

**Core Components of the Reflection API:**

The primary entry point for reflection in Java is the java.lang.Class class. An object of type Class provides methods to get information about a class or interface, such as its name, modifiers, fields, methods, constructors, and superclasses.

Some key classes and interfaces in the java.lang.reflect package (where most reflection-related functionalities reside) include:

* **Class:** Represents a class or interface. You can obtain a Class object in several ways:
  + Using the .class literal (e.g., String.class).
  + Calling the getClass() method on an object (e.g., "hello".getClass()).
  + Using the static Class.forName() method (e.g., Class.forName("java.lang.String")).
* **Field:** Represents a field (instance variable or static variable) of a class or interface.
* **Method:** Represents a method of a class or interface.
* **Constructor:** Represents a constructor of a class.
* **Modifier:** Provides static methods to interpret class and member access modifiers.
* **Array:** Provides static methods to dynamically create and access arrays.

**Common Uses of Reflection:**

* **Examining Class Structure:** Getting information about the fields, methods, constructors, and modifiers of a class.
* **Instantiating Objects:** Creating new objects of a class at runtime, even if the class name is not known at compile time.
* **Invoking Methods:** Calling methods of an object at runtime, even if the method name is not known at compile time.
* **Accessing and Modifying Fields:** Getting and setting the values of fields of an object at runtime, even private fields.
* **Working with Annotations:** Reading annotations associated with classes, methods, and fields at runtime (as we saw earlier).
* **Dynamic Proxy Creation:** Creating proxy objects that implement interfaces and can intercept method calls.
* **Framework Development:** Many Java frameworks (like Spring and Hibernate) heavily use reflection to discover and manage components, configure dependencies, and map objects to database tables.

**Examples of Reflection in Action:**

Let's look at some basic examples:

**1. Getting Class Information:**

```Java

public class ReflectionDemo {

public static void main(String[] args) {

Class<?> stringClass = String.class;

System.out.println("Class Name: " + stringClass.getName());

System.out.println("Simple Name: " + stringClass.getSimpleName());

System.out.println("Package Name: " + stringClass.getPackageName());

System.out.println("Is Interface: " + stringClass.isInterface());

System.out.println("Superclass: " + stringClass.getSuperclass().getName());

try {

Class<?> myClass = Class.forName("java.util.ArrayList");

System.out.println("Class Name (from name): " + myClass.getName());

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

```

**2. Getting Fields of a Class:**

```Java

import java.lang.reflect.Field;

class MyData {

public String publicField = "Public";

private int privateField = 123;

protected boolean protectedField = true;

}

public class FieldReflectionDemo {

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

MyData data = new MyData();

Class<?> dataClass = data.getClass();

Field[] publicFields = dataClass.getFields(); // Returns only public fields (inherited as well)

System.out.println("Public fields:");

for (Field field : publicFields) {

System.out.println(" " + field.getName() + ": " + field.get(data));

}

Field[] declaredFields = dataClass.getDeclaredFields(); // Returns all declared fields (public, private, protected)

System.out.println("\nDeclared fields:");

for (Field field : declaredFields) {

field.setAccessible(true); // Allows access to private fields

System.out.println(" " + field.getName() + ": " + field.get(data));

}

}

}

```

**3. Getting Methods of a Class:**

```Java

import java.lang.reflect.Method;

import java.util.Arrays;

class MyMethods {

public void method1() { System.out.println("Method 1"); }

private String method2(String arg) { return "Method 2 with " + arg; }

public int method3(int a, int b) { return a + b; }

}

public class MethodReflectionDemo {

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

MyMethods obj = new MyMethods();

Class<?> methodsClass = obj.getClass();

Method[] publicMethods = methodsClass.getMethods(); // Returns public methods (inherited as well)

System.out.println("Public methods:");

Arrays.stream(publicMethods).forEach(method -> System.out.println(" " + method.getName()));

Method[] declaredMethods = methodsClass.getDeclaredMethods(); // Returns all declared methods

System.out.println("\nDeclared methods:");

for (Method method : declaredMethods) {

System.out.println(" " + method.getName() + " with parameters: " + Arrays.toString(method.getParameterTypes()));

method.setAccessible(true); // Allow access to private methods

if (method.getName().equals("method2")) {

String result = (String) method.invoke(obj, "Reflection");

System.out.println(" Result of method2: " + result);

} else if (method.getName().equals("method3")) {

int sum = (int) method.invoke(obj, 5, 3);

System.out.println(" Result of method3: " + sum);

}

}

}

}

```

**4. Getting Constructors of a Class:**

```Java

import java.lang.reflect.Constructor;

import java.util.Arrays;

class MyConstructorClass {

public MyConstructorClass() { System.out.println("Default constructor"); }

public MyConstructorClass(String message) { System.out.println("Constructor with message: " + message); }

}

public class ConstructorReflectionDemo {

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

Class<?> constructorClass = MyConstructorClass.class;

Constructor<?>[] constructors = constructorClass.getConstructors(); // Returns public constructors

System.out.println("Public constructors:");

Arrays.stream(constructors).forEach(ctor -> System.out.println(" " + ctor.getName() + " with parameters: " + Arrays.toString(ctor.getParameterTypes())));

Constructor<?> defaultCtor = constructorClass.getDeclaredConstructor();

MyConstructorClass obj1 = (MyConstructorClass) defaultCtor.newInstance();

Constructor<?> stringCtor = constructorClass.getDeclaredConstructor(String.class);

MyConstructorClass obj2 = (MyConstructorClass) stringCtor.newInstance("Hello from reflection");

}

}

```

**Drawbacks of Reflection:**

While reflection is powerful, it also has some drawbacks:

* **Performance Overhead:** Reflection operations are generally slower than direct method calls or field access because they involve runtime resolution of types and members.
* **Security Restrictions:** In some security contexts (e.g., applets or security-managed environments), there might be restrictions on using reflection, especially to access private members.
* **Increased Complexity:** Code that uses reflection can be more complex and harder to read and debug compared to direct code.
* **Loss of Compile-time Type Safety:** Reflection bypasses compile-time type checks, which can lead to runtime errors (like ClassCastException or NoSuchMethodException) if used incorrectly.

**When to Use Reflection:**

Reflection is most useful in situations where you need to:

* Work with classes or members whose names are not known at compile time (e.g., in frameworks or plugins).
* Inspect the structure of objects at runtime (e.g., for serialization or debugging).
* Create generic code that needs to operate on different types of objects.
* Implement features like dependency injection or object-relational mapping (ORM).

In general, reflection should be used judiciously and only when necessary, as it can make code less efficient and harder to maintain.

**Dates and Times**

Alright, let's move on to **Working with Dates and Times** in Java. Handling dates and times correctly is crucial in many applications. Java has evolved its date and time API significantly over the years.

Before Java 8, the primary classes for date and time were in the java.util package: Date and Calendar. However, these classes had several issues, including:

* **Mutability:** Both Date and Calendar objects are mutable, which can lead to subtle bugs when they are shared.
* **Poor Design:** The APIs were not very intuitive or consistent. For example, months in Calendar were 0-indexed.
* **Lack of Clarity:** It wasn't always clear whether a Date object represented a point in time or a date and time.
* **Time Zone Handling:** Time zone handling was complex and error-prone.

**The java.time Package (Java 8 and Later):**

Java 8 introduced the java.time package, which provides a comprehensive and well-designed API for working with dates, times, instants, and durations. This new API is immutable, thread-safe, and much more intuitive.

**Key Classes in java.time:**

* **LocalDate:** Represents a date (year, month, day) without time or time-zone.

```Java

import java.time.LocalDate;

public class LocalDateDemo {

public static void main(String[] args) {

LocalDate today = LocalDate.now();

System.out.println("Today's date: " + today);

LocalDate specificDate = LocalDate.of(2023, 10, 26);

System.out.println("Specific date: " + specificDate);

int year = today.getYear();

java.time.Month month = today.getMonth();

int dayOfMonth = today.getDayOfMonth();

System.out.println("Year: " + year + ", Month: " + month + ", Day: " + dayOfMonth);

LocalDate tomorrow = today.plusDays(1);

LocalDate lastMonth = today.minusMonths(1);

System.out.println("Tomorrow: " + tomorrow);

System.out.println("Last month: " + lastMonth);

boolean isLeap = today.isLeapYear();

System.out.println("Is " + year + " a leap year? " + isLeap);

}

}

```

* **LocalTime:** Represents a time (hour, minute, second, nanosecond) without date or time-zone.

```Java

import java.time.LocalTime;

public class LocalTimeDemo {

public static void main(String[] args) {

LocalTime now = LocalTime.now();

System.out.println("Current time: " + now);

LocalTime specificTime = LocalTime.of(14, 30, 45);

System.out.println("Specific time: " + specificTime);

int hour = now.getHour();

int minute = now.getMinute();

int second = now.getSecond();

int nano = now.getNano();

System.out.println("Hour: " + hour + ", Minute: " + minute + ", Second: " + second + ", Nano: " + nano);

LocalTime later = now.plusHours(2).minusMinutes(30);

System.out.println("Later time: " + later);

}

}

```

* **LocalDateTime:** Represents a date-time without a time-zone. It's essentially a combination of LocalDate and LocalTime.

```Java

import java.time.LocalDateTime;

public class LocalDateTimeDemo {

public static void main(String[] args) {

LocalDateTime now = LocalDateTime.now();

System.out.println("Current date and time: " + now);

LocalDateTime specificDateTime = LocalDateTime.of(2024, java.time.Month.DECEMBER, 25, 9, 0, 0);

System.out.println("Specific date and time: " + specificDateTime);

LocalDate datePart = now.toLocalDate();

LocalTime timePart = now.toLocalTime();

System.out.println("Date part: " + datePart);

System.out.println("Time part: " + timePart);

LocalDateTime inFuture = now.plusDays(10).plusHours(5);

System.out.println("In future: " + inFuture);

}

}

```

* **Instant:** Represents a point in time on the UTC timeline. It's often used for machine-readable timestamps.

```Java

import java.time.Instant;

import java.time.temporal.ChronoUnit;

public class InstantDemo {

public static void main(String[] args) {

Instant now = Instant.now();

System.out.println("Current instant (UTC): " + now);

Instant specificInstant = Instant.ofEpochSecond(1672531200L); // Example: Jan 1, 2023 UTC

System.out.println("Specific instant: " + specificInstant);

Instant laterInstant = now.plus(5, ChronoUnit.HOURS);

System.out.println("Later instant: " + laterInstant);

long epochSeconds = now.getEpochSecond();

int nanoSeconds = now.getNano();

System.out.println("Epoch seconds: " + epochSeconds);

System.out.println("Nano seconds: " + nanoSeconds);

}

}

```

* **ZonedDateTime:** Represents a date-time with a time-zone. It's crucial for handling times in different geographical regions.

```Java

import java.time.LocalDateTime;

import java.time.ZoneId;

import java.time.ZonedDateTime;

public class ZonedDateTimeDemo {

public static void main(String[] args) {

LocalDateTime localDateTime = LocalDateTime.now();

ZoneId kolkataZone = ZoneId.of("Asia/Kolkata");

ZonedDateTime kolkataDateTime = ZonedDateTime.of(localDateTime, kolkataZone);

System.out.println("Kolkata time: " + kolkataDateTime);

ZoneId newYorkZone = ZoneId.of("America/New\_York");

ZonedDateTime newYorkDateTime = kolkataDateTime.withZoneSameInstant(newYorkZone);

System.out.println("New York time (same instant): " + newYorkDateTime);

System.out.println("Available Zone IDs: " + ZoneId.getAvailableZoneIds());

}

}

```

* **OffsetDateTime:** Represents a date-time with a time-zone offset from Greenwich/UTC.

```Java

import java.time.LocalDateTime;

import java.time.OffsetDateTime;

import java.time.ZoneOffset;

public class OffsetDateTimeDemo {

public static void main(String[] args) {

LocalDateTime localDateTime = LocalDateTime.now();

ZoneOffset offset = ZoneOffset.of("+05:30"); // India Standard Time offset

OffsetDateTime offsetDateTime = OffsetDateTime.of(localDateTime, offset);

System.out.println("Offset date time: " + offsetDateTime);

}

}

```

* **Duration:** Represents a time-based amount of time (e.g., "2 hours, 30 minutes").

```Java

import java.time.Duration;

import java.time.LocalDateTime;

public class DurationDemo {

public static void main(String[] args) {

LocalDateTime start = LocalDateTime.now();

LocalDateTime end = start.plusHours(3).plusMinutes(15);

Duration duration = Duration.between(start, end);

System.out.println("Duration: " + duration);

System.out.println("Duration in minutes: " + duration.toMinutes());

}

}

```

* **Period:** Represents a date-based amount of time (e.g., "2 years, 3 months, 10 days").

```Java

import java.time.LocalDate;

import java.time.Period;

public class PeriodDemo {

public static void main(String[] args) {

LocalDate date1 = LocalDate.of(2020, 1, 1);

LocalDate date2 = LocalDate.of(2023, 4, 11);

Period period = Period.between(date1, date2);

System.out.println("Period: " + period);

System.out.println("Years: " + period.getYears() + ", Months: " + period.getMonths() + ", Days: " + period.getDays());

}

}

```

**Formatting and Parsing Dates and Times:**

The java.time.format package provides classes for formatting and parsing date and time objects. The DateTimeFormatter class is the main class for this purpose.

```Java

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

import java.util.Locale;

public class DateTimeFormatterDemo {

public static void main(String[] args) {

LocalDateTime now = LocalDateTime.now();

// Predefined formats

DateTimeFormatter isoFormatter = DateTimeFormatter.ISO\_DATE\_TIME;

System.out.println("ISO format: " + now.format(isoFormatter));

// Custom format

DateTimeFormatter customFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");

System.out.println("Custom format: " + now.format(customFormatter));

// Parsing a string to LocalDateTime

String dateTimeString = "2024-05-18 11:00:00";

LocalDateTime parsedDateTime = LocalDateTime.parse(dateTimeString, customFormatter);

System.out.println("Parsed date time: " + parsedDateTime);

// Localized format

DateTimeFormatter localizedFormatter = DateTimeFormatter.ofLocalizedDateTime(java.time.format.FormatStyle.MEDIUM).withLocale(Locale.US);

System.out.println("Localized format (US): " + now.format(localizedFormatter));

}

}

```

**Interoperability with Legacy Date and Time Classes:**

The java.time package also provides methods to convert between the new date and time classes and the older java.util.Date and java.util.Calendar classes.

* Date.from(Instant instant): Converts an Instant to a Date.
* date.toInstant(): Converts a Date to an Instant.
* GregorianCalendar.from(ZonedDateTime zdt): Creates a GregorianCalendar from a ZonedDateTime.
* calendar.toZonedDateTime(): Converts a Calendar to a ZonedDateTime.

**Key Advantages of java.time:**

* **Immutability:** Objects are immutable, making them thread-safe and reducing the risk of unexpected changes.
* **Clearer API:** The API is more consistent and easier to understand.
* **Better Time Zone Handling:** The java.time.ZoneId and java.time.ZonedDateTime classes provide robust time zone support.
* **Separation of Concerns:** Clear distinction between dates, times, and date-times.

The java.time package is the recommended way to handle dates and times in Java 8 and later. It provides a much-improved API compared to the legacy classes.

**Localization and Internationalization**

Excellent! Let's move on to **Localization and Internationalization** in Java. When building applications that need to be used by a global audience, it's crucial to consider how to adapt them to different languages, regional customs, and cultural conventions. This is where localization (l10n) and internationalization (i18n) come into play.

**Internationalization (i18n):**

Internationalization is the process of designing and developing your application in a way that it can be adapted to various languages and regions without requiring engineering changes to the core code. It's about making your application globally ready.

**Localization (l10n):**

Localization is the process of adapting an internationalized application for a specific locale (a combination of language and region). This typically involves translating text, formatting dates, times, numbers, and currencies according to the conventions of the target locale.

**Key Concepts and Tools in Java for i18n and l10n:**

1. **Locales:** A Locale object in Java represents a specific geographical, political, or cultural region. It's identified by a language code and an optional country/region code (and sometimes a variant). You can create Locale objects using constants (e.g., Locale.US, Locale.FR), or using the Locale constructor with language and country codes (e.g., new Locale("en", "GB")).

```Java

import java.util.Locale;

public class LocaleDemo {

public static void main(String[] args) {

Locale usLocale = Locale.US;

Locale frenchLocale = Locale.FRANCE;

Locale germanGermany = Locale.GERMANY;

Locale englishUK = new Locale("en", "GB");

Locale malayalamIndia = new Locale("ml", "IN");

System.out.println("US Locale: " + usLocale.getDisplayCountry() + " - " + usLocale.getDisplayLanguage());

System.out.println("French Locale: " + frenchLocale.getDisplayCountry() + " - " + frenchLocale.getDisplayLanguage(frenchLocale));

System.out.println("German (Germany): " + germanGermany.getDisplayCountry(germanGermany) + " - " + germanGermany.getDisplayLanguage(germanGermany));

System.out.println("English (UK): " + englishUK.getDisplayCountry(englishUK) + " - " + englishUK.getDisplayLanguage(englishUK));

System.out.println("Malayalam (India): " + malayalamIndia.getDisplayCountry(malayalamIndia) + " - " + malayalamIndia.getDisplayLanguage(malayalamIndia));

Locale defaultLocale = Locale.getDefault();

System.out.println("Default Locale: " + defaultLocale.getDisplayCountry() + " - " + defaultLocale.getDisplayLanguage());

}

}

```

1. **Resource Bundles:** The primary way to manage locale-specific text and other resources in Java is through **Resource Bundles**. A resource bundle is a collection of locale-specific files (typically .properties files) that contain key-value pairs. The keys are the same across all locales, while the values are translated or adapted for each specific locale.
   * **Creating Resource Bundle Files:** You create .properties files with the base name followed by the locale identifier. For example, if your base name is messages, you might have:
     + messages.properties (default locale)
     + messages\_en.properties (English)
     + messages\_fr.properties (French)
     + messages\_de\_DE.properties (German - Germany)

**Example messages\_en.properties:**

Properties

greeting=Hello

farewell=Goodbye

welcome.message=Welcome, {0}! You have {1} new messages.

**Example messages\_fr.properties:**

Properties

greeting=Bonjour

farewell=Au revoir

welcome.message=Bienvenue, {0} ! Vous avez {1} nouveaux messages.

* + **Loading Resource Bundles:** You use the ResourceBundle.getBundle() method to load the appropriate bundle based on the current locale.

```Java

import java.util.Locale;

import java.util.ResourceBundle;

public class ResourceBundleDemo {

public static void main(String[] args) {

Locale currentLocale = Locale.getDefault(); // Or specify a locale: Locale.FRANCE;

ResourceBundle messages = ResourceBundle.getBundle("messages", currentLocale);

String greeting = messages.getString("greeting");

String farewell = messages.getString("farewell");

System.out.println("Greeting: " + greeting);

System.out.println("Farewell: " + farewell);

// Handling missing keys

try {

String unknown = messages.getString("unknown.key");

System.out.println("Unknown: " + unknown);

} catch (java.util.MissingResourceException e) {

System.err.println("Warning: Key not found - " + e.getKey());

}

}

}

```

\* \*\*Formatting Messages with Placeholders:\*\* You can use `java.text.MessageFormat` to handle messages with placeholders that can be replaced with dynamic values.

```java

import java.text.MessageFormat;

import java.util.Locale;

import java.util.ResourceBundle;

public class MessageFormatDemo {

public static void main(String[] args) {

Locale currentLocale = Locale.FRANCE;

ResourceBundle messages = ResourceBundle.getBundle("messages", currentLocale);

String welcomeMessagePattern = messages.getString("welcome.message");

String formattedMessage = MessageFormat.format(welcomeMessagePattern, "Jean", 3);

System.out.println(formattedMessage); // Output (for French locale): Bienvenue, Jean ! Vous avez 3 nouveaux messages.

currentLocale = Locale.US;

messages = ResourceBundle.getBundle("messages", currentLocale);

welcomeMessagePattern = messages.getString("welcome.message");

formattedMessage = MessageFormat.format(welcomeMessagePattern, "John", 5);

System.out.println(formattedMessage); // Output (for US locale): Welcome, John! You have 5 new messages.

}

}

```

1. **Formatting Dates and Times (using java.time.format.DateTimeFormatter):** As we discussed in the previous section, DateTimeFormatter allows you to format dates and times according to locale-specific patterns. You can use predefined localized formats or create custom ones.

```Java

import java.time.LocalDateTime;

import java.time.format.DateTimeFormatter;

import java.time.format.FormatStyle;

import java.util.Locale;

public class LocalizedDateTimeFormatDemo {

public static void main(String[] args) {

LocalDateTime now = LocalDateTime.now();

Locale usLocale = Locale.US;

DateTimeFormatter usFormatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL).withLocale(usLocale);

System.out.println("US Format: " + now.format(usFormatter));

Locale franceLocale = Locale.FRANCE;

DateTimeFormatter frenchFormatter = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.FULL).withLocale(franceLocale);

System.out.println("French Format: " + now.format(frenchFormatter));

}

}

```

1. **Formatting Numbers and Currencies (using java.text.NumberFormat and java.util.Currency):** The NumberFormat class provides locale-sensitive formatting for numbers, percentages, and currencies.

```Java

import java.text.NumberFormat;

import java.util.Currency;

import java.util.Locale;

public class NumberFormatDemo {

public static void main(String[] args) {

double amount = 12345.678;

Locale usLocale = Locale.US;

NumberFormat usNumberFormat = NumberFormat.getNumberInstance(usLocale);

System.out.println("US Number: " + usNumberFormat.format(amount));

NumberFormat usCurrencyFormat = NumberFormat.getCurrencyInstance(usLocale);

System.out.println("US Currency: " + usCurrencyFormat.format(amount));

Locale germanLocale = Locale.GERMANY;

NumberFormat germanNumberFormat = NumberFormat.getNumberInstance(germanLocale);

System.out.println("German Number: " + germanNumberFormat.format(amount));

NumberFormat germanCurrencyFormat = NumberFormat.getCurrencyInstance(germanLocale);

System.out.println("German Currency: " + germanCurrencyFormat.format(amount));

Currency euro = Currency.getInstance(germanLocale);

System.out.println("Euro Symbol: " + euro.getSymbol(germanLocale));

}

}

```

1. **Collation (using java.text.Collator):** Collation is the process of comparing and sorting strings in a locale-sensitive manner. The Collator class provides this functionality. Different locales have different rules for sorting (e.g., case sensitivity, handling of accented characters).

```Java

import java.text.Collator;

import java.util.Arrays;

import java.util.List;

import java.util.Locale;

public class CollatorDemo {

public static void main(String[] args) {

List<String> words = Arrays.asList("apple", "äpple", "banana", "zebra");

// Default sorting (based on Unicode values)

words.sort(String::compareTo);

System.out.println("Default sort: " + words);

// German sorting

Locale germanLocale = Locale.GERMANY;

Collator germanCollator = Collator.getInstance(germanLocale);

List<String> wordsGerman = Arrays.asList("apple", "äpple", "banana", "zebra");

wordsGerman.sort(germanCollator);

System.out.println("German sort: " + wordsGerman);

// US English sorting (case-insensitive)

Locale usLocale = Locale.US;

Collator usCollator = Collator.getInstance(usLocale);

usCollator.setStrength(Collator.PRIMARY); // Ignore case and accents

List<String> wordsCaseInsensitive = Arrays.asList("Apple", "apple", "Banana");

wordsCaseInsensitive.sort(usCollator);

System.out.println("US Case-insensitive sort: " + wordsCaseInsensitive);

}

}

```

**Best Practices for i18n and l10n:**

* **Separate Localizable Resources:** Keep all locale-specific text, images, and other resources separate from your core application logic (typically in resource bundles).
* **Use Unicode:** Ensure your application uses Unicode (UTF-8 is highly recommended) for encoding and handling text to support a wide range of characters.
* **Consider Cultural Conventions:** Be aware of differences in date and time formats, number formats, currency symbols, address formats, and other cultural conventions across locales.
* **Test Thoroughly:** Test your application in various locales to ensure that it is correctly localized and that the user experience is appropriate for each target audience.
* **Plan Early:** Internationalization should be considered early in the development process rather than as an afterthought. Retrofitting i18n can be significantly more complex.

By following these principles and using the appropriate Java APIs, you can build applications that are truly global and can effectively cater to users from different linguistic and cultural backgrounds.

**Lombok in Java**

**What is Project Lombok?**

Project Lombok is a popular Java library that aims to reduce boilerplate code. It does this by automatically generating various methods (like getters, setters, constructors, equals(), hashCode(), toString()) at compile time based on annotations you place in your source code. This means you write less code, making your classes more concise and readable.

**Why is Lombok Used?**

Java, while powerful, often requires a lot of repetitive, "boilerplate" code for simple data classes. For example, a typical Plain Old Java Object (POJO) representing a user might need:

* Private fields for id, name, email.
* Public getter methods for each field.
* Public setter methods for each field (if mutable).
* A constructor to initialize all fields.
* equals() and hashCode() methods to correctly compare objects.
* A toString() method for debugging and logging.

This leads to classes that are mostly boilerplate, obscuring the actual business logic or data they represent. Lombok steps in to remove this clutter, allowing developers to focus on the essential parts of their code.

**How Does Lombok Work?**

Lombok operates as an **annotation processor** during the compilation phase. When your Java source code is compiled, Lombok's annotation processor intercepts the compilation process. It reads the Lombok annotations (like @Getter, @Setter, @Data) and, based on these annotations, *generates* the corresponding bytecode directly into your compiled .class files.

Crucially, this means the generated methods are *not* visible in your .java source file, but they *are* present in the compiled .class file, making them available to the JVM and other parts of your application at runtime.

**Key Features and Popular Annotations with Examples:**

Let's look at some of the most commonly used Lombok annotations:

**1. @Getter and @Setter**

These annotations generate public getter and setter methods for the fields they annotate. You can place them at the field level or at the class level.

* **At field level:** Generates getter/setter only for that specific field.
* **At class level:** Generates getter/setter for all non-static fields in the class.

**Example:**

```Java

import lombok.Getter;

import lombok.Setter;

// User.java

public class User {

private int id; // No Lombok annotation, no generated getter/setter for id by default

@Getter @Setter // Generates public getName() and setName(String name)

private String name;

@Getter // Generates public getEmail()

private String email;

// We can still write custom methods if needed

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

// No setter for email, so it's read-only once constructed or explicitly set

public User(int id, String name, String email) {

this.id = id;

this.name = name;

this.email = email;

}

public static void main(String[] args) {

User user = new User(1, "Alice", "alice@example.com");

// Using generated getter/setter for 'name'

System.out.println("Initial Name: " + user.getName()); // Calls generated getName()

user.setName("Alicia"); // Calls generated setName()

System.out.println("Updated Name: " + user.getName());

// Using generated getter for 'email'

System.out.println("Email: " + user.getEmail()); // Calls generated getEmail()

// Using custom getter/setter for 'id'

System.out.println("ID: " + user.getId());

user.setId(2);

System.out.println("Updated ID: " + user.getId());

}

}

```

**2. @NoArgsConstructor and @AllArgsConstructor**

These annotations generate constructors.

* **@NoArgsConstructor:** Generates a public constructor with no arguments. Fields are initialized to their default values (0 for numbers, null for objects, false for booleans).
* **@AllArgsConstructor:** Generates a public constructor with arguments for all fields in the class, in the order they are declared.

**Example:**

```Java

import lombok.NoArgsConstructor;

import lombok.AllArgsConstructor;

// Product.java

@NoArgsConstructor // Generates: public Product() { }

@AllArgsConstructor // Generates: public Product(String name, double price, int stock) { ... }

public class Product {

private String name;

private double price;

private int stock;

// You still need getters/setters if you want to access/modify fields

// For simplicity, let's just add toString() here to see the object state

@Override

public String toString() {

return "Product{" +

"name='" + name + '\'' +

", price=" + price +

", stock=" + stock +

'}';

}

public static void main(String[] args) {

// Using @NoArgsConstructor

Product product1 = new Product();

System.out.println("Product 1 (NoArgsConstructor): " + product1); // Output: Product{name='null', price=0.0, stock=0}

// Using @AllArgsConstructor

Product product2 = new Product("Laptop", 1200.50, 50);

System.out.println("Product 2 (AllArgsConstructor): " + product2); // Output: Product{name='Laptop', price=1200.5, stock=50}

}

}

```

**Note:** If you use @AllArgsConstructor, and then also @NoArgsConstructor, and you have final fields, you must initialize final fields in the constructor. Lombok will correctly handle this for @AllArgsConstructor but for @NoArgsConstructor the final fields will not be initialized by the generated constructor, thus causing a compile error. This is why @NoArgsConstructor is often used with mutable fields or when combined with @Data (which makes fields non-final by default).

**3. @ToString, @EqualsAndHashCode**

* **@ToString:** Generates a toString() method that prints the class name and the values of all non-static fields.
* **@EqualsAndHashCode:** Generates equals(Object other) and hashCode() methods. These methods compare/hash all non-static fields by default.

**Example:**

```Java

import lombok.ToString;

import lombok.EqualsAndHashCode;

// Coordinates.java

@ToString // Generates toString()

@EqualsAndHashCode // Generates equals() and hashCode()

public class Coordinates {

private int x;

private int y;

private String description; // Will be included in toString, equals, hashCode

public Coordinates(int x, int y, String description) {

this.x = x;

this.y = y;

this.description = description;

}

public static void main(String[] args) {

Coordinates c1 = new Coordinates(10, 20, "Home");

Coordinates c2 = new Coordinates(10, 20, "Home");

Coordinates c3 = new Coordinates(30, 40, "Office");

// Using generated toString()

System.out.println("C1: " + c1); // Output: Coordinates(x=10, y=20, description=Home)

System.out.println("C3: " + c3); // Output: Coordinates(x=30, y=40, description=Office)

// Using generated equals()

System.out.println("C1 equals C2: " + c1.equals(c2)); // Output: C1 equals C2: true

System.out.println("C1 equals C3: " + c1.equals(c3)); // Output: C1 equals C3: false

// Using generated hashCode()

System.out.println("C1 hash code: " + c1.hashCode());

System.out.println("C2 hash code: " + c2.hashCode()); // Should be same as C1

}

}

```

**Important Note:** For @EqualsAndHashCode, you can use exclude or of parameters to specify which fields to include/exclude. For example, @EqualsAndHashCode(exclude = "description") would make c1 and c2 equal even if their descriptions differ, as long as x and y are the same.

**4. @Data**

This is a very powerful shortcut annotation that bundles several common annotations:

* @ToString
* @EqualsAndHashCode
* @Getter for all fields
* @Setter for all *non-final* fields
* @RequiredArgsConstructor (generates a constructor with arguments for all final fields and fields annotated with @NonNull).

It's excellent for typical POJOs or data classes.

**Example:**

```Java

import lombok.Data;

import lombok.NonNull;

// Book.java

@Data // Implies @ToString, @EqualsAndHashCode, @Getter, @Setter (for non-final), @RequiredArgsConstructor

public class Book {

@NonNull // This field will be part of the generated constructor by @Data's @RequiredArgsConstructor

private String title;

private String author;

private double price;

private final String isbn; // This final field will also be part of the constructor

// Lombok will generate:

// public Book(String title, String isbn) { this.title = title; this.isbn = isbn; }

// All getters (getTitle(), getAuthor(), getPrice(), getIsbn())

// Setters for non-final fields (setAuthor(), setPrice())

// toString(), equals(), hashCode()

public static void main(String[] args) {

// Using the constructor generated by @Data (due to @NonNull and final fields)

Book book = new Book("The Great Adventure", "978-1234567890");

book.setAuthor("Jane Doe"); // Using generated setter

book.setPrice(25.99); // Using generated setter

System.out.println("Book details: " + book);

// Output: Book(title=The Great Adventure, author=Jane Doe, price=25.99, isbn=978-1234567890)

Book book2 = new Book("The Great Adventure", "978-1234567890");

book2.setAuthor("Jane Doe");

book2.setPrice(25.99);

System.out.println("Books are equal: " + book.equals(book2)); // True (generated equals)

System.out.println("Book hash code: " + book.hashCode());

}

}

```

**5. @Builder**

This annotation generates a builder pattern for your class, which is especially useful for creating objects with many optional parameters or for immutable objects.

**Example:**

```Java

import lombok.Builder;

import lombok.Getter;

import lombok.ToString;

// Employee.java

@Builder // Generates an inner static Builder class and a build() method

@Getter // Generates getters for all fields

@ToString // Generates toString()

public class Employee {

private final int id;

private final String firstName;

private final String lastName;

private final String position;

private final double salary;

private final String department;

// Lombok generates a private constructor:

// private Employee(int id, String firstName, ...) { ... }

public static void main(String[] args) {

// Using the generated builder pattern

Employee employee1 = Employee.builder()

.id(101)

.firstName("John")

.lastName("Doe")

.position("Software Engineer")

.salary(75000.00)

.department("IT")

.build();

System.out.println("Employee 1: " + employee1);

// Output: Employee(id=101, firstName=John, lastName=Doe, position=Software Engineer, salary=75000.0, department=IT)

// Create another employee with fewer fields

Employee employee2 = Employee.builder()

.id(102)

.firstName("Jane")

.lastName("Smith")

.build(); // Position, salary, department will be null/default

System.out.println("Employee 2: " + employee2);

// Output: Employee(id=102, firstName=Jane, lastName=Smith, position=null, salary=0.0, department=null)

}

}

```

**6. @Slf4j (and other Logging Annotations)**

Lombok also provides annotations for integrating with common logging frameworks, automatically creating a logger instance.

* @Slf4j: For SLF4J (Simple Logging Facade for Java)
* @Log4j2: For Apache Log4j 2
* @CommonsLog: For Apache Commons Logging
* And others...

**Example:**

```Java

import lombok.extern.slf4j.Slf4j;

// MyService.java

@Slf4j // Generates: private static final org.slf4j.Logger log = org.slf4j.LoggerFactory.getLogger(MyService.class);

public class MyService {

public void performAction(String actionName) {

log.info("Performing action: {}", actionName); // Using the generated 'log' instance

try {

// Simulate some work

Thread.sleep(100);

log.debug("Action {} completed successfully.", actionName);

} catch (InterruptedException e) {

log.error("Action {} interrupted!", actionName, e); // Logging with exception

Thread.currentThread().interrupt();

}

}

public static void main(String[] args) {

MyService service = new MyService();

service.performAction("Database Cleanup");

service.performAction("Report Generation");

}

}

```

**To run this example, you would need SLF4J and Logback (or Log4j2) dependencies in your pom.xml or build.gradle:**

```XML

<dependencies>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.18.30</version> <scope>provided</scope>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.36</version> </dependency>

<dependency>

<groupId>ch.qos.logback</groupId>

<artifactId>logback-classic</artifactId>

<version>1.2.11</version> </dependency>

</dependencies>

```

**Pros of Using Lombok:**

1. **Reduced Boilerplate Code:** This is the primary and most significant advantage. It makes your source code much cleaner, shorter, and easier to read, as you don't see repetitive getters/setters, constructors, etc.
2. **Increased Readability:** Focusing on the essential fields and methods makes the class's purpose clearer.
3. **Faster Development:** Less typing means faster initial setup for data classes.
4. **Automatic Maintenance:** If you add or remove fields, Lombok automatically regenerates the associated methods, preventing manual errors and omissions.
5. **Consistency:** Ensures equals(), hashCode(), and toString() implementations are consistent across your data classes.

**Cons of Using Lombok:**

1. **IDE Plugin Requirement:** To avoid compilation errors and for proper IDE support (autocomplete, refactoring, error highlighting), you *must* install a Lombok plugin for your IDE (e.g., IntelliJ IDEA, Eclipse, VS Code). Without it, your IDE won't "see" the generated methods.
2. **Learning Curve:** New developers joining a project need to be aware of Lombok and how it works, as the methods they expect to see in the source code are not there.
3. **Debugging:** While generally not an issue, debugging tools might sometimes require specific configurations or show generated code in a less intuitive way.
4. **Dependency on a Third-Party Library:** Your build process now depends on Lombok. While it's very stable, it's still an external dependency.
5. **Hidden Code:** Some developers dislike the "magic" of generated code, preferring to see everything explicitly written out for full transparency. This can be a point of debate in teams.
6. **Potential for Misuse:** Overuse of @Data or other annotations can sometimes lead to classes that aren't truly data-only or have unintended side effects if not understood properly.

**When to Use Lombok:**

Lombok is widely used in many Java projects, especially with frameworks like Spring Boot, where data transfer objects (DTOs), entities, and simple POJOs are common. It's particularly beneficial for:

* **Data Models:** Entities, DTOs, value objects.
* **Web Application Controllers:** To define request bodies or response structures.
* **Any class primarily serving as a data container.**

**Conclusion:**

Project Lombok is a powerful and very popular tool in the Java ecosystem for reducing boilerplate code. By understanding how it works as an annotation processor and familiarizing yourself with its common annotations, you can significantly streamline your Java development, make your code more concise, and improve its readability. While it introduces a dependency and requires IDE plugin setup, its benefits often outweigh these considerations for many projects