JAVA COMPLETE NOTES

FOR LAST-MINUTE INTERVIEW PREP

CONTAINS:

- Complete Notes
- Code Snippet's
- Interview tips
- Resources
- Lot more..

BG
Byte Genius

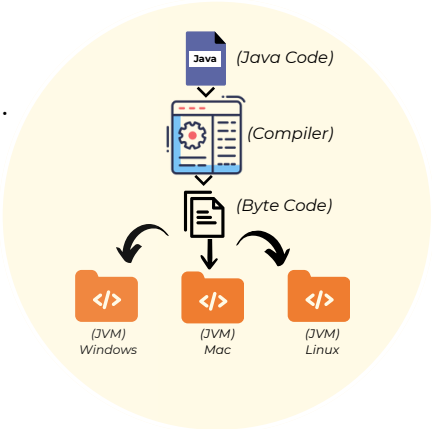
History & Importance

Father of Java

- Java was developed by **James Gosling** and his team at **Sun Microsystems** (later acquired by Oracle Corporation) in the mid-1990s.
- It was initially designed as a programming language for consumer electronic devices, such as set-top boxes and home appliances.

KEY FEATURES:

Platform Independence:



- Java introduced the concept of ***"write once, run anywhere"*** (WORA).
- Java programs can be compiled into bytecode, which is a platform-independent format.
- This bytecode can then be executed on any device or operating system with a Java Virtual Machine (JVM), making Java highly portable.

Object-Oriented Programming (OOP):

- Java was built from the ground up as an object-oriented language.
- OOP allows developers to write ***modular, reusable, and maintainable code*** by representing concepts as objects with properties (***attributes***) and behaviors (***methods***).
- This promotes code organization, flexibility, and extensibility.

Automatic Memory Management:

- Java introduced automatic memory management through its ***garbage collection mechanism***.
- Developers do not have to manage memory explicitly, as the ***JVM*** automatically deallocates memory occupied by objects that are no longer in use.

Rich Standard Library:

- Java provides a comprehensive standard library that offers a wide range of **pre-built classes** and **APIs.**
- This library includes utilities for ***I/O operations, networking, threading, database connectivity, and graphical user interfaces (GUIs),*** among others.

Robustness and Security:

- Java prioritizes reliability and security.
- It incorporates strong **error-checking mechanisms** during compilation and runtime, which helps catch errors early and ensures code stability.
- Java's security features, such as ***sandboxing, secure class loading, and byte-code verification***, make it suitable for building secure applications and applets.

WRITING YOUR FIRST "HELLO, WORLD!" PROGRAM!!

Code it Yourself

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

Let's break down the program:

- The program starts with the public class Hello World. In Java, each program resides within a class, and the class name must match the filename (in this case, HelloWorld.java).
- Inside the class, we have a **method** called **main**. This is the entry point for the program, where the execution begins.
- It has the following signature: **public static void main(String[] args)**. Don't worry about the details of this signature for now; we'll cover it in later chapters.
- Within the main method, we have a single line of code: **System.out.println("Hello, World!");.** This line prints the text "Hello, World!" to the console.
- **The System.out.println** function is used to display output in Java, and it automatically adds a **newline** after the text.

VARIABLES:

- In Java, variables are used to **store** and **manipulate data.**
- They act as named containers that ***hold values of a particular type***.
- Before using a variable, you need to declare it by specifying its type and name.

Code:

```
int age; // Declaration of an integer variable named "age"
```

Types of Variable:

In Java, variables can be classified into different types based on their usage and behavior.
Here are the main types of variables:

Local Variables:

- Local variables are declared within a **method, constructor, or block and have block or method scope.**
- They are only accessible within the **specific block** or **method** in which they are declared.
- Local variables must be **initialized** before they are used.

Example:

Code:

```
class FunWithJava {
  public void myMethod() {
 int localVar = 5; // Local variable
 System.out.println(localVar);
 }
}
```

- In this example, localVar is a local variable declared within the **myMethod()** method.
- It is only accessible within that **method**.

Instance Variables:

- Instance variables, also known as **member variables** or **fields**, are declared within a class but outside of any **method**, **constructor**, or **block**.
- They have class scope and are associated with an **instance (object)** of the class.
- Each instance of the class has its own set of instance variables.
- Instance variables are **automatically initialized with default values** if not **explicitly initialized.**

Example:

Code:

```
class FunWithJava {
  int instanceVar; // Instance variable
  public void myMethod() {
  instanceVar = 10; // Accessible within methods of the class
  }
}
```


- In this example, instanceVar is an **instance variable** of the MyClass class.
- Each instance of MyClass will have its own **instanceVar** variable.

Class Variables (Static Variables):

- Class variables, also known as **static variables**, are declared within a class and are preceded by the **static keyword**.
- They have class scope and are associated with the class itself rather than with **any particular instance** of the class.
- Class variables are shared among all instances of the class and can be accessed directly using the class name.
- Class variables are **initialized with default values** if not **explicitly initialized**.

Example:

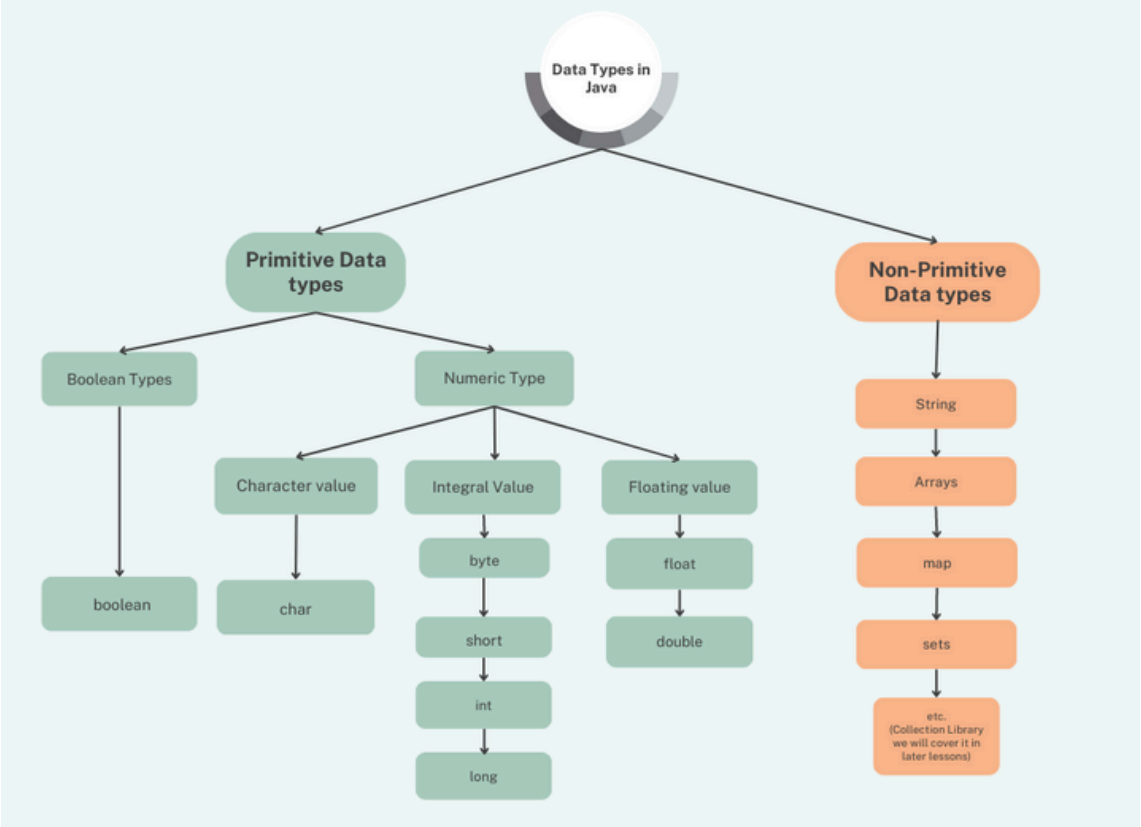
Code:

```
class FunWithJava {
  static int instanceVar; // Instance variable
     public void myMethod() {
     instanceVar = 10; // Accessible within methods of the class
     }
}
```

- In this example, classVar is a **class variable of the MyClass class**.
- All instances of MyClass will share the same **classVar variable**.

DATA TYPES:

- Data types in Java are of different **sizes** and **values** that can be stored in the variable that is made as per **convenience** and **circumstances** to cover up all test cases.



Primitive Data Types:

Primitive data types are the **basic building blocks of data** in Java.
They are predefined by the language and have **fixed sizes.**

The following are the primitive data types in Java:

- **byte**: 8-bit signed integer. It can hold values from ***-128 to 127.***
- **short**: 16-bit signed integer. It can hold values from ***-32,768 to 32,767.***
- **int**: 32-bit signed integer. It can hold values from ***-2,147,483,648 to 2,147,483,647.***
- **long**: 64-bit signed integer. It can hold larger whole numbers by adding an "L" suffix to the value ***(e.g., 1000000L).***
- **float**: 32-bit floating-point number. It can hold decimal numbers and requires an "f" or "F" suffix ***(e.g., 3.14f).***
- **double**: 64-bit floating-point number. It can hold larger decimal numbers and is the default type for decimal values ***(e.g., 3.14).***
- **boolean**: Represents a boolean value, either **true** or **false**.
- **char**: Represents a single Unicode character and is enclosed in single quotes ***(e.g., 'A', 'b', '#').***

Syntax:

```
byte byteVar = 2;
short shortVar = 45;
int intVar = 1000;
```

Non-Primitive Data Types/Reference Types:

Reference data types are **more complex** and are based on **classes or interfaces** defined by the programmer or provided by ***Java's standard library***(Collection Library).
They can hold references **(memory addresses)** to objects, *strings*, *arrays*, or other complex data structures.

The two commonly used reference data types are:

String:

- Represents a sequence of characters.
- It is a class in Java and provides various methods for manipulating strings.

Arrays:

- Represents a collection of elements of the same type.
- Arrays have a fixed length and provide methods for accessing and manipulating their elements.

It's important to note that the size of primitive data types remains the same on all platforms, while the size of reference data types may vary based on the platform and JVM implementation.

Syntax:

```
// Declare String without using new operator
String s = "GeeksforGeeks";

// Declare String using new operator
String s1 = new String("GeeksforGeeks");
```

OPERATORS:

Operators are symbols or special characters that perform operations on operands (variables, literals, or expressions).
Java supports various types of operators that allow you to perform mathematical, logical, bitwise, and other operations.

Arithmetic Operators:

+ (addition): Adds two operands.
- (subtraction): Subtracts one operand from another.
*** (multiplication):** Multiplies two operands.
/ (division): Divides one operand by another.
% (modulus): Returns the remainder of the division.

Assignment Operators:

= (assignment): Assigns a value to a variable.
+=, -=, *=, /=, %= (compound assignment): Performs an operation and assigns the result to the variable.

Increment and Decrement Operators:

++ (increment): Increases the value of the operand by 1.
-- (decrement): Decreases the value of the operand by 1.

Bitwise Operators:

& (bitwise AND): Performs bitwise AND operation on operands.
| (bitwise OR): Performs bitwise OR operation on operands.
^ (bitwise XOR): Performs bitwise XOR operation on operands.
~ (bitwise complement): Flips the bits of the operand.

Logical Operators:

&& (logical AND): Returns true if both conditions are true.
|| (logical OR): Returns true if at least one condition is true.
! (logical NOT): Negates the boolean value.

Comparison Operators:

== (equal to): Checks if two operands are equal.
!= (not equal to): Checks if two operands are not equal.
>, < (greater than, less than): Checks if one operand is greater/less than the other.
>=, <= (greater than or equal to, less than or equal to): Checks if one operand is greater/less than or equal to the other.

Shift Operators:

>> (right shift): Shifts the bits of the operand to the right.
<< (left shift): Shifts the bits of the operand to the left.
>>> (unsigned right shift): Shifts the bits of the operand to the right, filling the empty positions with zeros.

Conditional Operators:

?: (ternary operator): Evaluates a condition and returns one of two values based on the result.

Checkout out the official docs from Java

Example for Operators:

Code it Yourself:

```
public class OperatorExample {
    public static void main(String[] args) {
        // Arithmetic Operators
        int a = 10;
        int b = 3;
        int result;

        result = a + b; // Addition
        System.out.println("Addition: " + result);

        // Relational Operators
        boolean isEqual = a == b; // Equal to
        System.out.println("Equal to: " + isEqual);

        // Logical Operators
        boolean logicalAnd = isEqual && (a > 0); // Logical AND
        System.out.println("Logical AND: " + logicalAnd);

        // Assignment Operators
        int c = 5;
        c += 3; // Equivalent to: c = c + 3
        System.out.println("After 'c += 3', c = " + c);

        // Bitwise Operators
        int x = 5; // Binary: 0101
        int y = 3; // Binary: 0011

        int bitwiseAnd = x & y; // Bitwise AND: 0001 (Decimal: 1)
        System.out.println("Bitwise AND: " + bitwiseAnd);

        // Shift Operators
        int num = 8; // Binary: 1000

        int leftShifted = num << 2;
        // Left Shift: 0010 0000 (Decimal: 32)
        System.out.println("Left Shift: " + leftShifted);
    }
}
```

INPUT AND OUTPUT IN JAVA:

In Java, **input and output (I/O) operations** allow you to **interact** with **the user** and **the external environment**.
Java provides **several classes and methods** to handle input and output operations **efficiently**.

Here's an explanation of input and output in Java:

Input in Java:

Input **refers** to **the process of accepting data** or **information from the user** or **an external source** and **making it available for processing in a Java program**.

Some common ways to perform input operations in Java include:

Reading from the Console:

- Java provides the **Scanner class**, which allows you **to read user input** from the **console**.
- You can use methods like **nextInt()**, **nextLine()**, or **nextDouble()** to read different types of input.

Example:

Code:

```
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");
        scanner.close();
    }
}
```

Different methods to take input via Scanner():

Method	Description
nextBoolean()	Used for reading Boolean value
nextByte()	Used for reading Byte value
nextDouble()	Used for reading Double value
nextFloat()	Used for reading Float value
nextInt()	Used for reading Int value
nextLine()	Used for reading Line value
nextLong()	Used for reading Long value
nextShort()	Used for reading Short value

Writing to the Console:

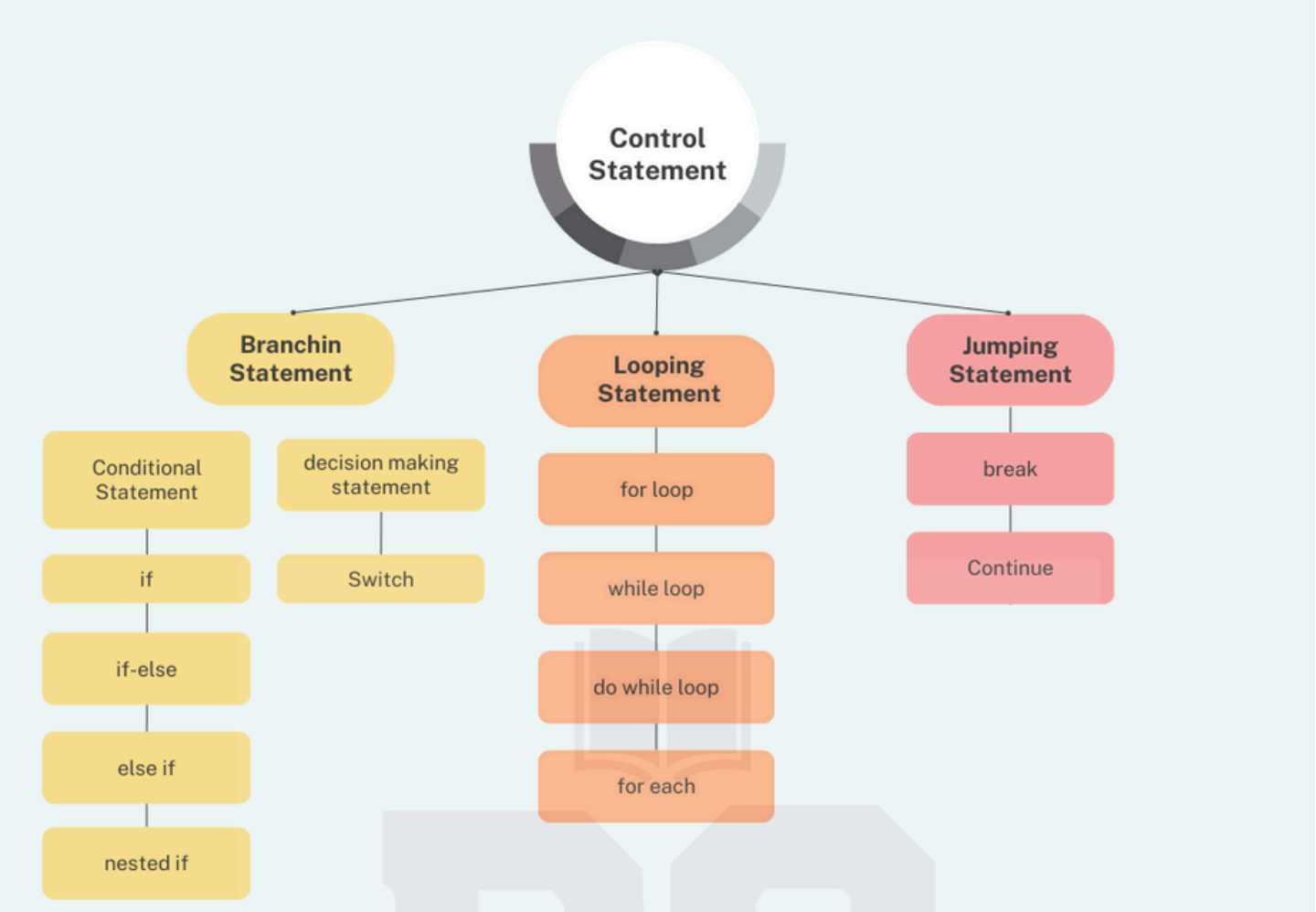
- The most straightforward way to display output in Java is by using the **System.out object** and its **println() method**.

Example:

Code:

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

FLOW CONTROL IN JAVA:



- Flow control in Java refers to **the management and direction of program execution** based on **conditions** and **loops**.
- It determines the order in which **statements** and **blocks of code are executed**, **allowing you to make decisions**, **repeat actions**, and **control the flow of the program's execution**.

Conditional Statements(if-else,switch):

if-else statement:

- Allows you to execute a block of code if a condition is true, and an alternative block of code if the condition is false.

Syntax:

Syntax:

```
if (condition) {
    // Executed if condition is true
} else {
    Executed if condition is false
}
```

Example:

Code:

```
int age = 18;

if (age >= 18) {
    System.out.println("Eligible to vote!");
} else {
    System.out.println("Not eligible to vote!"
}
```

Switch statement:

- Provides a way to **select one of many code blocks to be executed** based on the **value of a variable** or **an expression**.

Syntax:

```
switch (variable) {
    case value1:
        // Code block executed if variable matches value1
        break;
    case value2:
        // Code block executed if variable matches value2
        break;
    // ...
    default:
        // Code block executed if variable doesn't match any case
        break;
}
```

Example:

Code:

```
int dayOfWeek = 3;

switch (dayOfWeek) {
    case 1:
        System.out.println("It's Monday");
        break;
    case 2:
        System.out.println("It's Tuesday");
        break;
    case 3:
        System.out.println("It's Wednesday");
        break;
    default:
        System.out.println("It's another day of the week");
        break;
}
```

Loops (For, While, do-While):

- In Java, loops are used to **repeat a block of code multiple times**.
- They allow you to **automate repetitive tasks**, **iterate over data structures**, and **control the flow of execution**.

Some common ways to perform input operations in Java include:

for Loop:

- The for loop is commonly used when you know **the exact number of iterations** required.
- It consists of three parts: **initialization**, **condition**, and **iteration expression**.
- The **initialization** is executed **before the loop starts**, the **condition** is checked **before each iteration**, and the **iteration** expression is executed **at the end of each iteration**.

Syntax:

Syntax:

```
for (initialization; condition; iteration) {
    // Code block to be repeated
}
```

Example:

Code:

```
for (int i = 0; i < 5; i++) {
    System.out.println("Iteration: " + i);
}
```

while Loop:

- The while loop is used when you want to **repeat a block of code** as long as a **condition is true**.
- It checks the condition **before each iteration**.
- If the condition is **true**, the loop **body is executed**.
- If the condition is **false**, the loop is **exited**.

Syntax:

Syntax:

```
while (condition) {
    // Code block to be
repeated
}
```

Example:

Code:

```
for (int i = 0; i < 5; i++) {
    System.out.println(
"Iteration: " + i);
}
```

do-while Loop:

- The do-while loop is **similar to the while loop** but guarantees that the **loop body is executed at least once**.

- It checks the condition at **the end of each iteration**.
- If the condition is **true**, the **loop continues.**
- If the condition is **false**, the **loop is exited**.

Syntax:

Syntax:

```
do {
    // Code block to be
repeated
} while (condition);
```

In this example, the loop will execute five times, printing the current number.

Example:

Code:

```
int num = 1;

do {
    System.out.println("Number: " +
num);
    num++;
} while (num <= 5);
```

Control Transfer Statements (continue,break):

- Control transfer statements in Java allow you to **alter the flow of program execution**.
- They provide mechanisms to **transfer control** to **different parts of your code** based on **specific conditions or requirements**.
- Java supports several control transfer statements: **break**, **continue**, **return**, and **throw**.

Let's explain each of them:

break:

- The break statement is used to **terminate the current loop** or **switch statement** and **transfer control to the next statement** after the loop or switch.
- It allows you to **exit a loop prematurely** or **break out of a switch statement**.

Example:

Code:

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;  // Terminate the loop when i is 5
    }
    System.out.println(i);
}
```

NOTE:

In this example, when i reaches the value 5, the break statement is encountered, and the loop is exited.
The output will be 0 1 2 3 4.

continue:

- The continue statement is used to **skip** the remaining code in the current iteration of a loop and move to the next iteration.
- It allows you to **skip specific iterations** or **conditions** within a loop.

Example:

Code:

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 0) {
        continue;  // Skip even numbers
    }
    System.out.println(i);
}
```

NOTE:

In this example, the continue statement is encountered when it is an even number. It skips the remaining code in that iteration and moves to the next iteration.
The output will be 1 3 5 7 9.

return:

- The return statement is used to **exit a method and return a value to the caller**.
- It terminates the **execution of the current method and passes control back to the code** that invoked it.

Example:

Code:

```
public int add(int a, int b) {
    return a + b;  // Return the sum of a and b
}
```

NOTE:

In this example, the return statement is used to exit the add method and return the sum of a and b to the caller.

throw:

- The throw statement is used to **explicitly throw an exception**.
- It is used when you want **to handle exceptional conditions** and indicate an **abnormal state in your program**.

Example:

Code:

```
public int divide(int a, int b) {
    if (b == 0) {
        throw new ArithmeticException("Division by
zero is not allowed");
    }
    return a / b;
}
```

NOTE:

In this example, if b is 0, the throw statement is encountered, and it throws an ArithmeticException with a custom error message.

STRINGS IN JAVA:

- In Java, strings are **a sequence of characters** used to represent text.
- They are represented by **the String class** and are **widely used for storing and manipulating textual data**.
- Strings in Java are **immutable**, meaning **their values cannot be changed once they are created**.

Here's an explanation of strings in Java:

Declaring and Initializing Strings:

- You can **declare a string variable** by specifying **the type String followed by the variable name.**
- Strings can be **initialized in several ways:**

Code:

```
String message = "Hello, World!";  // Initializing with a string literal
String name = new String("John");  // Initializing with a new String object
String emptyString = "";           // Initializing with an empty string
```

Methods:

String Concatenation:

- In Java, you can concatenate (**join**) strings together using the **+ operator**.
- This operation creates a new string that combines the contents of the concatenated strings.

Example:

Code:

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;  // Concatenating strings
```

String Length:

- The **length()** method of the String class allows you to obtain the length (**number of characters**) of a string.

Example:

Code:

```
String text = "Hello";
int length = text.length();  // Returns the length of the string
(5 in this case)
```

String Comparison:

- To compare strings for equality, you should use the **equals()** method, which checks if *two strings have the same sequence* of characters.

Example:

Code:

```
String str1 = "Hello";
String str2 = "Hello";
boolean isEqual = str1.equals(str2);  // Returns true since the strings have the same value
```

String Manipulation:

- The String class provides various methods for **manipulating** and **extracting** information from strings.
- Some commonly used methods include **toUpperCase(), toLowerCase(), substring(), replace(), split(), and charAt().**
- These methods allow you to perform tasks such as **converting case, extracting substrings, replacing characters, splitting strings into arrays**, and **accessing individual characters.**

Example:

Code:

```
String text = "Hello, World!";
String upperCaseText = text.toUpperCase();
// Converts the string to uppercase
String substring = text.substring(7, 12);
// Extracts the substring "World"
String replacedText = text.replace("Hello", "Hi");
// Replaces "Hello" with "Hi"
String[] splitArray = text.split(",");
// Splits the string into an array ["Hello", " World!"]
char firstChar = text.charAt(0);
// Accesses the first character 'H'
```

String Formatting:

- Java provides the String.format() method and the printf() method (from System.out) to format strings by replacing placeholders with actual values based on specific patterns. This allows you to create formatted output.

Code:

```
int age = 25;
System.out.printf("I am %d years old", age);
```

ARRAYS IN JAVA:

- In Java, an array is a data structure that allows you to **store multiple values** of the same type in a **contiguous memory block.**
- Arrays provide a convenient way to work with **collections of data**, such as a list of numbers or a set of objects.

Here's an explanation of Arrays in Java:

Declaring and Creating Arrays:

- You can declare an array variable by *specifying the type of the elements* followed by **square brackets []**, along with the variable name.
- Arrays can be created using the **new** keyword, specifying the **size or length** of the array.

Example:

Code:

```
int[] numbers;                      // Declaring an array variable
numbers = new int[5];               // Creating an array of size 5

// Alternatively, you can declare and create an array in one line:
int[] numbers = new int[5];
```

Accessing Array Elements:

- Array elements are accessed using their index, which **starts from 0** and goes up to **length - 1.**
- You can **access** and **modify** individual elements of an array using the index enclosed in **square brackets [].**

Example:

Code:

```
int[] numbers = {1, 2, 3, 4, 5};    // Initializing an array
int firstNumber = numbers[0];       // Accessing the first element (index 0)
numbers[2] = 10;                    // Modifying the value of the third element
(index 2)
```

Array Length:

- The length of an array represents the **number of elements** it can hold.
- You can retrieve the length of an array using the **length property**.

Code:

```
int[] numbers = {1, 2, 3, 4, 5};
int length = numbers.length;        // length of the array (5 in this case)
```

Iterating over Arrays:

- You can use **loops**, such as the for loop, to iterate over the elements of an array.
- By utilizing the array's length, you can **access each element sequentially.**

Example:

Code:

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

for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);   // Accessing each element of the array
}
```

Array Initialization:

- Arrays can be initialized during **declaration** or later using the **assignment operator =..**

Example:

Code:

```
int[] numbers = {1, 2, 3, 4, 5};    // Initializing during declaration

int[] numbers = new int[5];         // Creating an empty array
numbers[0] = 10;                    // Initializing individual elements
numbers[1] = 20;
// ...
```

Multi-dimensional Arrays:

- Java also supports multi-dimensional arrays, such as **2D arrays** or **arrays of arrays.**
- These are useful for representing matrices or tables of data.

Example:

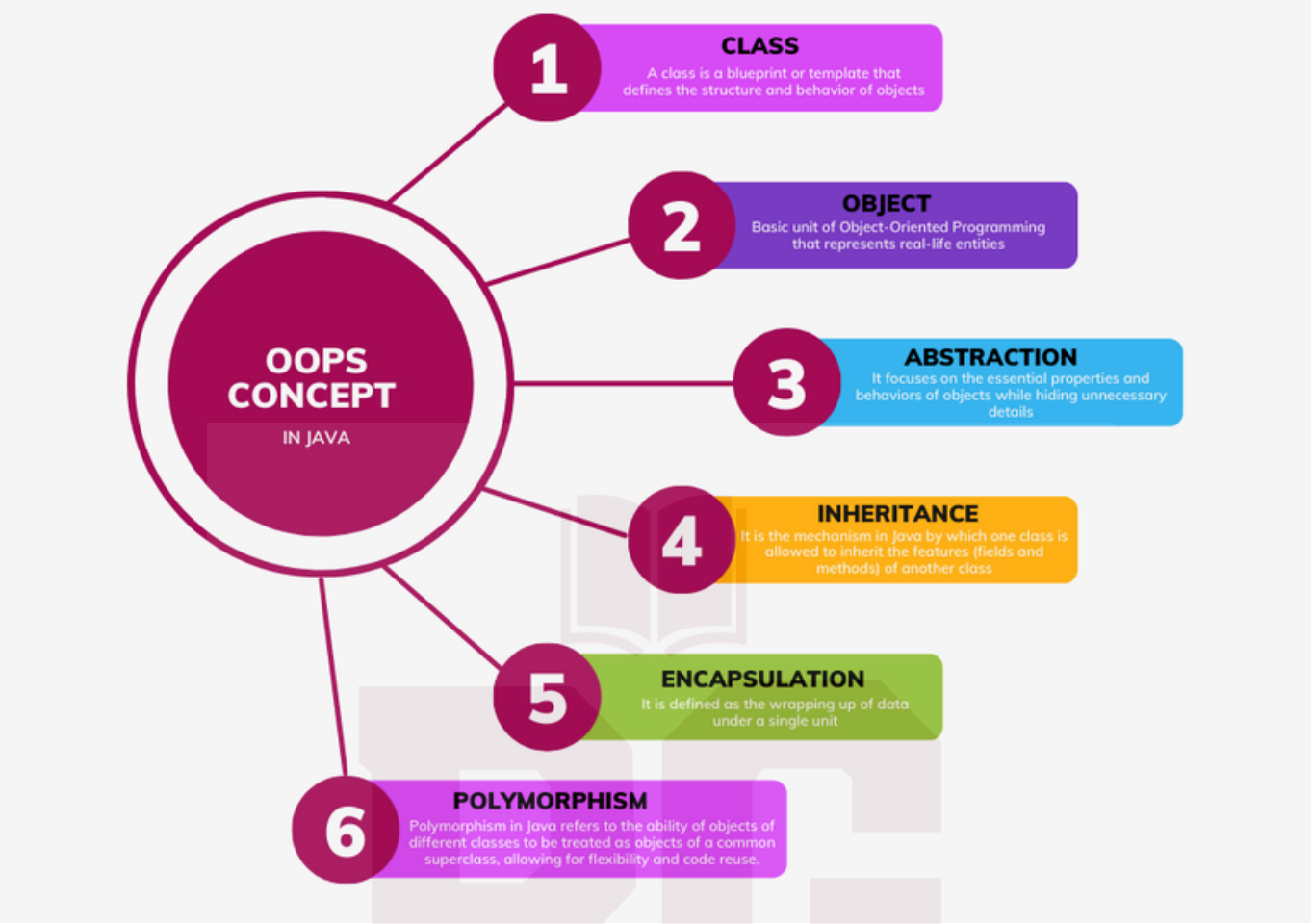
Code:

```
int[][] matrix = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};

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

For more refer this Link:

OOPS IN JAVA:



OOps overview: Pillars

- Object-oriented programming (OOP) in Java is a **programming paradigm** that revolves around **the concept of objects**.
- It is a powerful and widely-used approach for **designing** and **implementing software**.
- OOP in Java is based on four fundamental principles: **encapsulation, inheritance, polymorphism,** and **abstraction**.

Let's explain each principle briefly:

- Java supports these **OOP principles** with its **class-based inheritance model**.
- You can **create classes**, **define attributes** and **methods**, **instantiate objects**, and **interact with them using references** and **method calls**.
- Java's extensive standard library provides **built-in classes** and **interfaces that exemplify the principles of OOP** and **facilitate building robust** and **modular applications**.
- By adhering to OOP principles, you can write **maintainable**, **reusable**, and **efficient code in Java**, **making it easier to manage complex systems** and **collaborate with other developers**.

- Object-Oriented Programming (OOP) is like **playing with Lego blocks**.
- Imagine you have different types of Lego blocks, **each with its own shape and color**.
- In OOP, these **Lego blocks** are like **"classes".**

Classes:

- A class is like **a blueprint** or **a template for creating objects**.
- It defines **what an object can do (methods)** and **what data it can store (attributes)**.
- **For example**, a "Car" class can have **methods** like **"start"**, **"accelerate"**, and **"stop"**, as well as **attributes** like **"color"** and **"fuel"**.

Objects:

- An object is like a **real Lego structure built** using **a specific Lego block (class)**.
- It represents **a single instance of that class**.
- **For example**, a "Car" object represents a specific car with its unique color and fuel level.

Encapsulation:

- Encapsulation is like **keeping the inner workings** of a **Lego structure hidden** inside, so others **can't easily mess it up**.
- In OOP, we use **access modifiers** (public, private, etc.) **to control what parts of a class are accessible** to other classes.
- This ensures that **data is protected** and **used in a controlled manner**.

Inheritance:

- Inheritance is like **creating new Lego structures using existing structures as a base**.
- It allows one class to **inherit the attributes** and **methods** of another class.
- **For example**, you can have a "SportsCar" class that inherits from the "Car" class and adds specific methods or attributes unique to sports cars.

Polymorphism:

- Polymorphism is like **being able to use different types of Lego structures interchangeably**.
- In OOP, it allows you to treat **objects of different classes** as if they were of the same type.
- **For example**, you can have a method that works with both "Car" objects and "SportsCar" objects, even though they are different classes.

Abstraction:

- Abstraction is like **creating Lego structures using only the necessary details and hiding the rest**.
- In OOP, it's about **creating classes** with only the **essential methods** and attributes **visible to other classes**, **hiding the complexity** and **unnecessary details**.

- Overall, OOP allows you to **create organized, modular,** and **reusable code by representing real-world concepts as classes** and **interacting with them through objects**.
- It's a powerful way to build **complex programs by breaking them into smaller**, **manageable parts**, just like **building amazing structures with Lego blocks!**

Introduction to classes and objects:

Classes:

- A class is a **blueprint or a template for creating objects**.
- It defines **the structure and behavior of objects of that type**.
- It encapsulates **data (attributes)** and **methods (functions)** that operate on that data. In simple terms, a class is like **a user-defined data type that you can use to create objects**.

Here's an example of a simple class in Java:

Example:

Code:

```
public class Dog {
    // Attributes
    String name;
    int age;
    // Method
    public void bark() {
        System.out.println("Woof! Woof!");
    }
}
```

NOTE:

In this example, we have defined a class called "Dog" with two attributes name and age, and a method bark().

Objects:

- An object is **an instance of a class**.
- It represents **a specific entity or concept from the real world that you want to model in your program**.
- Objects are **created based on the blueprint** defined by the class.
- You can c**reate multiple objects from the same class**, and each object will have **its own set of attribute values**.

Here's how you create objects from the "Dog" class:

Example:

Code:

```
public class Main {
    public static void main(String[] args) {
        // Creating objects
        Dog dog1 = new Dog();
        Dog dog2 = new Dog();

        // Setting attributes
        dog1.name = "Buddy";
        dog1.age = 3;

        dog2.name = "Max";
        dog2.age = 5;

        // Calling methods
        dog1.bark();  // Output: Woof! Woof!
        dog2.bark();  // Output: Woof! Woof!
    }
}
```

NOTE:

In this example, we created two objects, dog1 and dog2, from the "Dog" class. We set their attributes name and age, and then called the bark() method on both objects.

- **Using classes and objects,** you can **model complex systems** by representing their **components as classes** and **creating instances (objects)** to interact with these components.
- This allows you to **write modular, reusable, and organized code, making it easier to build** and **maintain large-scale applications**.

Pillar 1 : Abstraction

- Data abstraction is one of the four fundamental pillars of object-oriented programming (OOP) in Java.
- It involves the concept of simplifying complex systems by representing only the essential features and hiding unnecessary details from the user.
- For beginners, data abstraction can be compared to everyday objects or devices that have buttons and functionalities without revealing their internal workings.

Let's explain data abstraction in Java:

Abstraction in the Real World:

- In our daily lives, we interact with many objects or devices that have interfaces (buttons, knobs, screens) to perform specific functions.
- We don't need to know how the internal mechanisms work; we only need to know how to use them effectively.
- For example, when driving a car, we use the pedals and steering wheel without worrying about the engine's internal functioning.

Example:

Let's consider a simple example of a "Car" class that represents a car's basic functionalities without revealing the internal mechanics:

Code:

```
public class Car {
    // Attributes (data)
    private String make;
    private String model;
    private int year;

    // Constructor
    public Car(String make, String model, int year) {
        this.make = make;
        this.model = model;
        this.year = year;
    }

    // Method to start the car
    public void start() {
        System.out.println("Car started.");
    }

    // Method to accelerate the car
    public void accelerate() {
        System.out.println("Car is accelerating.");
    }

    // Method to stop the car
    public void stop() {
        System.out.println("Car stopped.");
    }
}
```

NOTE:

In this example, the "Car" class has attributes make, model, and year, and methods start(), accelerate(), and stop(). The internal details of how these methods work are hidden from external code.

Using Abstraction:

Example:

Now, any external code can create objects of the "Car" class and use its methods without knowing how those methods are implemented:

Code:

```
public class Main {
    public static void main(String[] args) {
        Car myCar = new Car("Toyota", "Camry",
2022);
        myCar.start();
        myCar.accelerate();
        myCar.stop();
    }
}
```

NOTE:

In this example, the "Car" class has attributes make, model, and year, and methods start(), accelerate(), and stop(). The internal details of how these methods work are hidden from external code.

- By using data abstraction, you can create classes that are easy to use, understand, and maintain.
- It simplifies the programming process and allows developers to focus on the essential aspects of their code, making it more organized and less error-prone.

Pillar 2: Encapsulation

- Data encapsulation is one of the four fundamental pillars of object-oriented programming (OOP) in Java.
- It refers to the practice of bundling data (attributes) and methods (behavior) that operate on that data together within a class, and controlling access to that data from outside the class.
- For beginners, data encapsulation can be compared to a gift wrapped in a box: the gift's content is hidden and can only be accessed through the wrapping.

Let's explain data encapsulation in Java:

Encapsulation in the Real World:

- In everyday life, we use objects that have certain functionalities.
- For example, a smartphone has buttons, a touchscreen, and apps, but we don't need to know the inner workings of the phone's hardware or software to use it.
- This is similar to data encapsulation, where we hide the internal details of a class from the user, presenting only a well-defined interface to interact with it.

Encapsulation in Java:

- In Java, data encapsulation is achieved by using access modifiers (public, private, protected, and default).
- These modifiers control the visibility and accessibility of class members (attributes and methods) from outside the class.
- By marking attributes as private and providing public methods to access or modify them, we encapsulate the data and ensure that it is accessed in a controlled manner.

Private Attributes and Public Methods:

- In Java, it is common to declare attributes as private, which means they can only be accessed within the same class.
- To interact with these private attributes, we create public methods (getters and setters) that allow external code to read or modify the attributes.
- This way, the internal data is hidden, and any modifications are done through the methods, ensuring data integrity.

Example:

Let's consider a simple example of a "Person" class with a private attribute "age" and public methods "getAge()" and "setAge()" to encapsulate the data:

Code:

```
public class Person {
    private int age; // private attribute

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

    // public setter method
    public void setAge(int newAge) {
        if (newAge >= 0) {
            age = newAge;
        }
    }
}
```

NOTE:

In this example, the "age" attribute is marked as private, and its value can only be accessed or modified using the getAge() and setAge() methods.

Using Encapsulation:

Example:

External code can create objects of the "Person" class and use the public methods to interact with the "age" attribute:

Code:

```
public class Main {
    public static void main(String[] args)
{
        Person person = new Person();
        person.setAge(25); // set the age

        int age = person.getAge();
// get the age
        System.out.println("Age: " + age);
// Output: Age: 25
    }
}
```

NOTE:

In this example, this code creates a "Person" object, sets the age using the public method setAge(), and retrieves the age using the public method getAge().

- By using data encapsulation, you ensure that the internal state of objects is protected, preventing unintended access or modifications. It makes your code more reliable, maintainable, and secure.
- Data encapsulation is an essential concept in OOP, and practicing it helps build well-organized and robust Java programs.

Pillar 3: Inheritance

- Data inheritance, also known as class inheritance, is one of the four fundamental pillars of object-oriented programming (OOP) in Java.
- It allows a class (subclass or derived class) to inherit properties and behaviors from another class (superclass or base class).
- For beginners, data inheritance can be compared to a family tree, where children inherit certain traits and characteristics from their parents.

Let's explain data inheritance in Java:

Inheritance in the Real World:

- In our daily lives, we often observe inheritance in families.
- Children inherit certain physical traits, characteristics, and behaviors from their parents.
- For example, a child may inherit the eye color, height, and hair type from their mother and the personality traits and skills from their father.

Inheritance in Java:

- In Java, class inheritance allows one class (subclass) to inherit the attributes and methods of another class (superclass).
- The subclass can extend the functionality of the superclass, add its own attributes and methods, and override inherited methods.
- This promotes code reuse, as common features can be defined in a superclass and reused in multiple subclasses.

Extending a Class:

- To establish inheritance in Java, you use the extends keyword to indicate that a class is derived from another class.
- The subclass follows the superclass's structure and can add new features or modify existing ones.

Example:

Let's consider a simple example with a superclass "Animal" and a subclass "Dog":

Code:

```
// Superclass
public class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

// Subclass
public class Dog extends Animal {
    public void bark() {
        System.out.println("Dog is barking.");
    }
}
```

NOTE:

In this example, the "Dog" class extends the "Animal" class using the extends keyword. The "Dog" class inherits the eat() method from the "Animal" class.

Using Inheritance:

Example:

Now, we can create objects of the "Dog" class and use the inherited method eat():

Code:

```
public class Main {
    public static void main(String[] args)
{
        Dog dog = new Dog();
        dog.eat();  // Output: Animal is
eating.
        dog.bark(); // Output: Dog is
barking.
    }
}
```

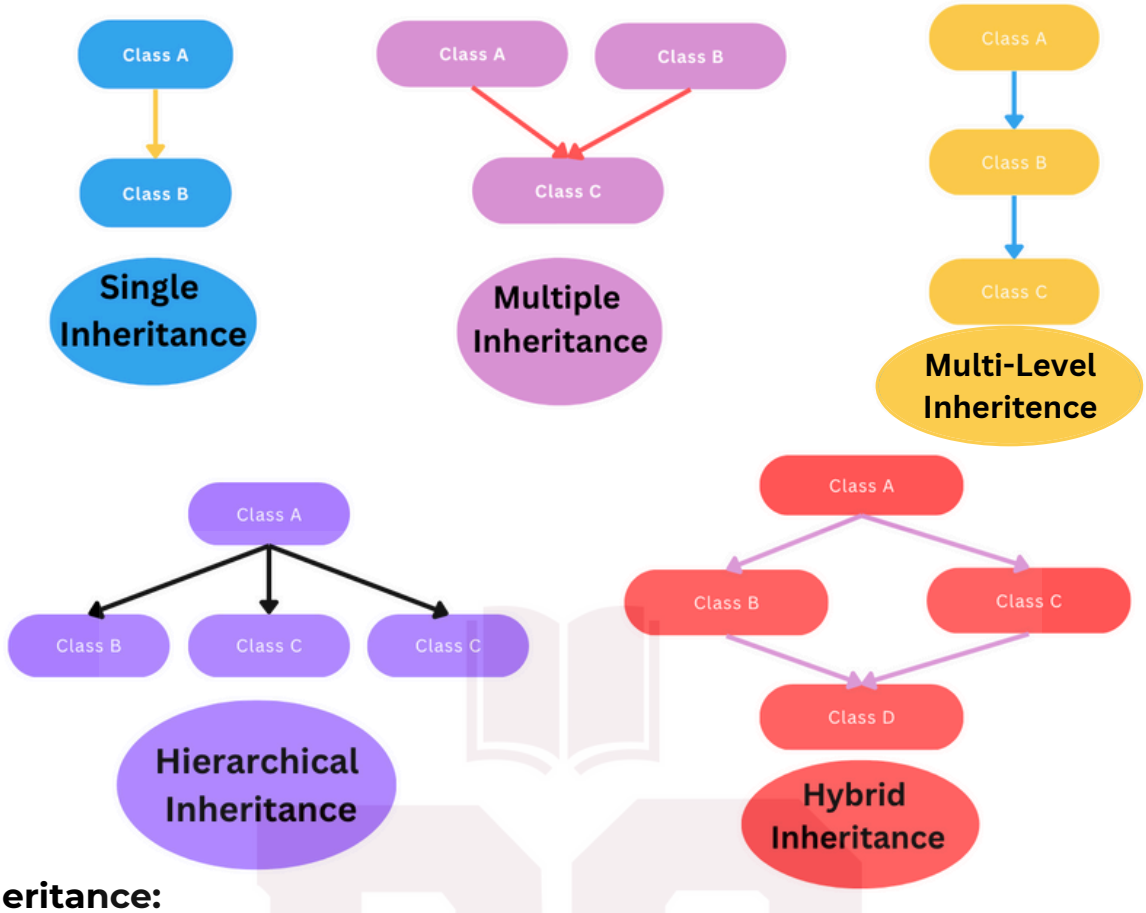
NOTE:

This code creates a "Dog" object and calls both the inherited method eat() from the "Animal" class and the method bark() from the "Dog" class.

- By using data inheritance in Java, you can create a hierarchical relationship among classes, organize code more efficiently, and promote code reuse.
- This leads to more manageable and modular programs. Inheritance is a powerful mechanism in OOP that enables you to model real-world relationships and build complex systems with ease.

Types of Inheritance:

- **Single Inheritance**
- **Multilevel Inheritance**
- **Hierarchical Inheritance**
- **Multiple Inheritance**
- **Hybrid Inheritance**



Single Inheritance:

- Single inheritance involves one class inheriting from a single superclass.
- Java supports single inheritance, where a class can extend only one superclass.
- This ensures a clear and straightforward relationship between the classes in the inheritance hierarchy.

Example:

Code:

```
// Superclass
public class Animal {
    // attributes and methods
}

// Subclass
public class Dog extends Animal {
    // additional attributes and methods
}
```

Multiple Inheritance (Through Interfaces):

- Multiple inheritance allows a class to inherit from multiple superclasses.
- However, Java does not support multiple inheritance directly for classes to avoid ambiguity and diamond problems.
- Java achieves multiple inheritance through interfaces, where a class can implement multiple interfaces, effectively inheriting their abstract methods.

Example:

Code:

```
// First Interface
public interface CanRun {
    void run();
}

// Second Interface
public interface CanSwim {
    void swim();
}

// Implementing Multiple Interfaces
public class Duck implements CanRun, CanSwim {
    @Override
    public void run() {
        // implementation
    }

    @Override
    public void swim() {
        // implementation
    }
}
```

Multilevel Inheritance:

- Multilevel inheritance involves a chain of inheritance, where a subclass becomes the superclass for another subclass.
- This creates a multilevel relationship between classes, leading to a linear inheritance hierarchy.

Example:

Code:

```
// Superclass
public class Animal {
    // attributes and methods
}

// First Subclass
public class Mammal extends Animal {
    // additional attributes and methods
}

// Second Subclass
public class Dog extends Mammal {
    // additional attributes and methods
}
```

- Java's inheritance mechanism provides a powerful way to build class hierarchies and promote code reuse.
- It helps organize classes, abstract common behaviors, and create more manageable and modular programs.

Pillar 4: Polymorphism

- Data Polymorphism is one of the four fundamental pillars of object-oriented programming (OOP) in Java.
- It refers to the ability of objects of different classes to be treated as objects of a common superclass.
- For beginners, polymorphism can be compared to using a single remote control to operate multiple types of electronic devices, such as a TV, DVD player, or sound system.

Let's explain polymorphism in Java:

Polymorphism in the Real World:

- In our daily lives, we encounter polymorphism when dealing with various objects that share common features.
- For example, you can turn on/off different electronic devices using a universal remote control because they all have a common power button, even though they are different devices.

Polymorphism in Java:

- In Java, polymorphism allows you to use a reference of a superclass type to refer to objects of different subclasses.
- This means you can treat objects of different classes as if they were objects of a common superclass.
- This enhances flexibility and code reuse in your programs.

Polymorphic References:

- When you declare a reference variable of a superclass type and assign an object of a subclass to it, you achieve polymorphism.
- The reference variable can be used to call methods that are defined in the superclass and overridden in the subclass.

Example:

Let's consider a simple example with a superclass "Animal" and two subclasses "Dog" and "Cat":

Code:

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

// Subclass 1
public class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}

// Subclass 2
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows.");
    }
}
```

NOTE:

In this example, both "Dog" and "Cat" classes extend the "Animal" class and override the makeSound() method.

Using Polymorphism:

Example:

Now, we can use polymorphism to create objects and call the overridden method using the superclass reference:

Code:

```
public class Main {
 public static void main(String[] args) {
 Dog dog = new Dog();
 dog.eat(); // Output: Animal is eating.
 dog.bark(); // Output: Dog is barking.
 }
}
```

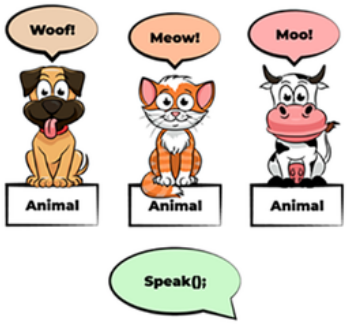
NOTE:

This code creates "Dog" and "Cat" objects and assigns them to "Animal" references animal1 and animal2. When calling the makeSound() method, the overridden version in each subclass is executed based on the actual object's type.

- By using polymorphism in Java, you can write flexible and generalized code that works with different types of objects without being aware of their specific implementations.
- This makes your code more adaptable, easier to maintain, and allows you to handle varying situations in a unified manner.
- Polymorphism is a powerful concept in OOP that contributes to the reusability and extensibility of your Java programs.

Types of Polymorphism:

- **Compile-time polymorphism (also known as method overloading)**
- **Run-time polymorphism (also known as method overriding)**



These two types allow you to write flexible and adaptable code by using a single interface to represent different behaviors or by providing different implementations based on the context.

Let's explain each type of polymorphism in Java:

Compile-Time Polymorphism:

- also known as method overloading, is a concept in Java where you can define multiple methods in a class with the same name but different parameter lists.
- The Java compiler determines which method to call based on the number or types of arguments passed during the method call.
- It allows you to perform similar operations with different data types or argument combinations in a concise and readable way.

Let's explain method overloading in Java with a beginner-friendly real-world example:

Real World Example: Calculator

Imagine you have a calculator that can perform addition. In our simplified version of the calculator, it can add integers, floating-point numbers, and even concatenate strings. Instead of creating different methods with different names for each type of addition, we can use method overloading.

Code:

```
public class Calculator {
    // Method for integer addition
    public int add(int a, int b) {
        return a + b;
    }

    // Method for double addition
    public double add(double a, double b) {
        return a + b;
    }

    // Method for string concatenation
    public String add(String str1, String str2) {
        return str1 + str2;
    }
}
```

NOTE:

In this example, we have defined three add() methods in the "Calculator" class:

- add(int a, int b) - This method performs addition for two integer values and returns an integer result.
- add(double a, double b) - This method performs addition for two double values and returns a double result.
- add(String str1, String str2) - This method concatenates two strings and returns a new string result.

Example:

Now, let's use the Calculator class in our Main class:

Code:

```
public class Main {
    public static void main(String[] args) {
        Calculator calculator = new Calculator();

        int result1 = calculator.add(5, 10);
        System.out.println("Integer Addition: " + result1);
// Output: Integer Addition: 15

        double result2 = calculator.add(3.5, 2.5);
        System.out.println("Double Addition: " + result2);
// Output: Double Addition: 6.0

        String result3 = calculator.add("Hello, ", "Java!");
        System.out.println("String Concatenation: " +
result3); // Output: String Concatenation: Hello, Java!
    }
}
```

NOTE:

Output:
Integer Addition: 15
Double Addition: 6.0
String Concatenation: Hello, Java!

As you can see, we used the same method name **add**() for different types of addition, but the Java compiler automatically determines which version of the method to call based on the arguments passed.

This makes the code more readable and allows us to handle different data types using the same method name.

- **Compile-Time Polymorphism** with method overloading is a powerful feature that simplifies code and improves code reusability.
- It enables you to write cleaner and more expressive programs in Java.

Run-Time Polymorphism:

- also known as **method overriding**, is a concept in Java where a **subclass** provides a specific implementation of a method that is already defined in its **superclass.**
- The subclass can override the method to customize its behavior while maintaining the method **signature (name and parameters)** from the superclass.
- This allows you to treat different objects uniformly through their common superclass while executing their specific behaviors at **run-time.**

Let's explain method overriding in Java with a beginner-friendly real-world example:

Real-World Example: Animals and Their Sounds

Imagine you have a program that represents different animals and their sounds. We'll create a superclass called "Animal" that has a method called makeSound(), which will be overridden by specific animal subclasses.

Code:

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

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

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

NOTE:

In this example, we have defined three classes: Animal, Dog, and Cat. The Animal class serves as the superclass, while Dog and Cat are subclasses that inherit from the Animal class.

- The Animal class has a method called makeSound() that prints "Animal makes a sound."
- The Dog class overrides the makeSound() method and provides a specific implementation, which prints "Dog barks."
- The Cat class also overrides the makeSound() method with its own implementation, which prints "Cat meows."

Example:

Now, let's use these classes in our Main class:

Code:

```
public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Animal();
        Animal animal2 = new Dog();
        Animal animal3 = new Cat();

        animal1.makeSound(); // Output: Animal makes a sound.
        animal2.makeSound(); // Output: Dog barks.
        animal3.makeSound(); // Output: Cat meows.
    }
}
```

As you can see, we created objects of both the Dog and Cat classes and assigned them to Animal references (animal2 and animal3).
Despite being referred to by their superclass type, when we call the *makeSound()* method on these objects, Java dynamically determines the appropriate implementation based on their actual types.

This is **run-time polymorphism** in action.

- Run-Time Polymorphism with method overriding allows you to achieve **flexibility** and **adaptability** in your Java programs.
- It lets you use a common **interface (the superclass)** to work with different objects (subclasses) while executing their specific behaviors based on their actual types at run-time.
- This promotes code **reusability** and simplifies the handling of various related classes within your application.

ACCESS MODIFIERS IN JAVA:

- Access modifiers in Java are keywords used to specify the **visibility** or accessibility of classes, methods, and variables within a Java program.
- They control which parts of the code can be **accessed** or **modified** by other classes or codes outside the current class.

There are four main access modifiers in Java:

Public:
The public access modifier allows **unrestricted** access to the class, method, or variable from any other part of the program, including code in other classes.

Private:
The private access modifier **restricts** access to the class, method, or variable only within the same class. It hides the implementation details from other classes, ensuring data encapsulation and information **hiding**.

Protected:
The protected access modifier allows access within the same package and by subclasses, even if they are in different packages. It provides a higher level of visibility than private but **more restricted** than public.

Default (Package-Private):
If no access modifier is specified, Java assigns the "default" access modifier. It allows access only within the same package but not from other packages.

Additionally, the class has three methods:

1. **displayInfo()**
2. **repair**
3. **start()**

Each with different access modifiers:

- **displayInfo()** is declared as public, allowing access from any part of the program.
- **repair()** is declared as private, restricting access to only within the same Car class.
- **start()** is declared as protected, allowing access within the same package and by subclasses, even if they are in different packages. By using appropriate access modifiers, you can control the visibility and accessibility of your class members, ensuring data encapsulation, information hiding, and proper encapsulation of behavior in your Java programs.

Real-World Example: Car Class

Let's use a real-world example of a Car class to illustrate access modifiers:

Code:

```
public class Car {
    public String make;           // Public attribute
    private String model;         // Private attribute
    protected int year;           // Protected attribute
    double price;                 // Default (Package-Private) attribute

    public Car(String make, String model, int year, double price) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.price = price;
    }

    public void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
        System.out.println("Price: " + price);
    }

    private void repair() {
        System.out.println("Car is being repaired.");
    }

    protected void start() {
        System.out.println("Car is starting.");
    }
}
```

NOTE:

In this example, we have a Car class with four attributes: make, model, year, and price. Each attribute has a different access modifier:

- make is declared as public, so it can be accessed from any part of the program.
- model is declared as private, so it can only be accessed within the same Car class.
- year is declared as protected, so it can be accessed within the same package and by subclasses, even if they are in different packages.
- price does not have an access modifier, making it "default" (package-private), so it can be accessed only within the same package.

Important Points:

- **If other programmers use your class, try to use the most restrictive access level that makes sense for a particular member.**
- **Use private unless you have a good reason not to.**
- **Avoid public fields except for constants.**

Access Modifier and their Visibility

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

CONSTRUCTORS IN JAVA:

- Constructors in Java are special methods used to initialize objects of a class.
- They are called automatically when an object is created using the new keyword.
- Constructors have the same name as the class and do not have a return type, not even void.
- They are used to set initial values to the object's attributes or perform any other necessary setup.

Real-World Example: Creating a Car Object

Let's use a real-world example of a Car class to illustrate constructors:

Code:

```
public class Car {
    // Attributes
    private String make;
    private String model;
    private int year;
    private double price;

    // Constructor
    public Car(String make, String model, int
year, double price) {
        this.make = make;
        this.model = model;
        this.year = year;
        this.price = price;
    }

    // Method to display car information
    public void displayInfo() {
        System.out.println("Make: " + make);
        System.out.println("Model: " + model);
        System.out.println("Year: " + year);
        System.out.println("Price: " + price);
    }
}
```

NOTE:

In this example, we have a Car class with four attributes: make, model, year, and price. We also have a constructor for the Car class that takes four parameters: make, model, year, and price. The constructor is used to initialize the attributes of the Car object with the values provided during object creation.

Now, let's create a Car object using the constructor:

Code:

```
public class Main {
    public static void main(String[] args) {
        // Creating a Car object using the constructor
        Car myCar = new Car("Toyota", "Camry", 2022, 25000.00);

        // Calling the displayInfo() method to show car
information
        myCar.displayInfo();
    }
}
```

When you run the Main class, it will create a Car object called myCar and initialize its attributes using the constructor. The displayInfo() method is then called to show the car's information on the console:

Code:

```
Make: Toyota
Model: Camry
Year: 2022
Price: 25000.0
```

In Java, there are three types of constructors based on their usage and purpose:

Default Constructor:

- A default constructor is **automatically** provided by Java if no constructor is **explicitly** defined in a class.
- It has *no parameters* and takes *no arguments.*
- Its main purpose is to initialize the object's attributes to their default values (e.g., 0 for numeric types, null for object references).
- It is used when an object is created without providing any specific initialization values.

Default Constructor:

- A default constructor is **automatically** provided by Java if no constructor is explicitly defined in a class.
- It has no parameters and takes no arguments.
- Its main purpose is to initialize the object's attributes to their default values (e.g., 0 for numeric types, null for object references).
- It is used when an object is created without providing any **specific initialization** values.
- If you define any other constructor in the class, Java will not provide the default constructor, and you need to define it explicitly if you still want to use it.

Example:

Code:

```
public class MyClass {
    // Default constructor automatically provided by Java
}
```

Parameterized Constructor:

- A parameterized constructor is defined with one or more parameters in the class.
- It allows you to provide specific values to initialize the object's attributes during object creation.
- The parameterized constructor is explicitly defined by the programmer to handle specific initialization requirements.

Example:

Code:

```
public class Car {
    private String make;
    private String model;

    // Parameterized constructor
    public Car(String make, String model) {
        this.make = make;
        this.model = model;
    }
}
```

Copy Constructor:

- A copy constructor is used to **create a new object** by copying the attributes of an existing object of the same class.
- It is useful when you want to create a **deep copy** (copying the content, not just the reference) of an object to avoid unintentional modifications.
- The copy constructor takes an object of the same class as a **parameter.**

Example:

Code:

```
public class Person {
    private String name;
    private int age;

    // Copy constructor
    public Person(Person otherPerson) {
        this.name = otherPerson.name;
        this.age = otherPerson.age;
    }
}
```

- Remember that a class can have **multiple constructors**, **including a mix of default**, **parameterized**, and **copy constructors.**
- The appropriate constructor is called based on the number and types of arguments passed during object creation.
- Constructors play a **crucial role** in object initialization and allow you to create objects with **specific initial values, enhancing the flexibility** and usability of Java programs.

OVERRIDE ANNOTATION IN JAVA:

- In Java, the **@Override** keyword is an annotation used to indicate that a method in a subclass is intended to override a method with the same **signature (name, return type, and parameters)** in its superclass.
- It is not mandatory to use the @Override annotation when overriding a method, but it is considered a good practice to include it.

The **@Override** annotation serves two main purposes:

Compile-Time Checking:

- When you use the **@Override** annotation, the Java compiler checks if the method with the same signature exists in the **superclass.**
- If it does not find such a method, it raises a **compilation error**, indicating that the method in the subclass is not correctly overriding any method from its superclass.

Code Documentation and Readability:

- Including the **@Override** annotation in the code helps improve its *readability and documentation.*
- When someone reads the code, they can immediately identify that the method in the subclass is meant to override a method from the superclass, **making the intent clearer.**

Example without @Override:

Code:

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

class Dog extends Animal {
    // Overriding the makeSound() method from the superclass
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}
```

Example with @Override:

Code:

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

class Dog extends Animal {
    // Using the @Override annotation to indicate method
overriding
    @Override
    public void makeSound() {
        System.out.println("Dog barks.");
    }
}
```

- In both examples, the Dog class overrides the **makeSound()** method from the Animal class.
- However, using the **@Override** annotation explicitly shows the intention of overriding, provides **compile-time checks**, and helps prevent accidental mistakes.
- It is worth noting that using **@Override** is not limited to only **superclass methods;** it can also be used when implementing methods from **interfaces**.
- Additionally, when a method is marked as **@Override,** it must have the same method signature as the method it is intending to override, ensuring that the method in the subclass truly overrides the one in the superclass or implements the one from the interface.

ABSTRACT CLASSES IN JAVA:

- An abstract class in Java is a class that cannot be instantiated on **its own and serves as a blueprint** for other classes.
- It can have abstract methods **(methods without implementation)** and concrete methods **(methods with implementation).**
- Abstract classes are used to define common behavior and attributes for related classes, providing a **template** for **subclasses** to follow.
- They allow you to **achieve abstraction** and create a **hierarchy** of classes with shared characteristics.

Key points about abstract classes in Java:

- An abstract class is declared using the abstract keyword before the class keyword.
- Abstract classes may have abstract methods (**methods without a body**) that must be implemented by their subclasses.
- Abstract classes may also have concrete methods (**methods with a body**) that are inherited by their **subclasses**.
- Abstract classes cannot be **instantiated directly**; they are meant to be subclassed.
- **Subclasses** that extend an abstract class must provide concrete implementations for all the abstract methods inherited from the superclass, or they themselves must be marked as abstract.

Real-World Example: Shape Hierarchy

Let's consider a real-world example of a shape hierarchy to illustrate the concept of an abstract class:

Code:

```
// Abstract class: Shape
abstract class Shape {
    // Abstract method: area (to be implemented by subclasses)
    public abstract double area();

    // Concrete method: displayInfo (common implementation for all shapes)
    public void displayInfo() {
        System.out.println("This is a shape.");
    }
}

// Subclass: Circle (extends Shape)
class Circle extends Shape {
    // Attributes
    private double radius;

    // Constructor
    public Circle(double radius) {
        this.radius = radius;
    }

    // Implementing the abstract method 'area' for Circle
    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}

// Subclass: Rectangle (extends Shape)
class Rectangle extends Shape {
    // Attributes
    private double length;
    private double width;
```

NOTE:

In this example, we have an abstract class Shape, which defines the common behavior for different shapes. It has one abstract method area() and one concrete method displayInfo(). The Circle and Rectangle classes are subclasses of the Shape class. They extend the Shape class and provide specific implementations for the area() method.

Continued

Code:

```
// Constructor
 public Rectangle(double length, double width) {
 this.length = length;
 this.width = width;
 }

 // Implementing the abstract method 'area' for Rectangle
 @Override
 public double area() {
 return length * width;
 }
}
```

Now, let's use these classes in the Main class:

Code:

```
public class Main {
    public static void main(String[] args) {
        Shape circle = new Circle(5.0);
        Shape rectangle = new Rectangle(4.0, 6.0);

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

        rectangle.displayInfo();
        System.out.println("Area of Rectangle: " + rectangle.area());
    }
}
```

Output:

```
This is a shape.
Area of Circle: 78.53981633974483
This is a shape.
Area of Rectangle: 24.0
```

- As you can see, we cannot create an object of the Shape class directly because it is **abstract.** However, we can create objects of the **concrete subclasses** Circle and Rectangle.
- The abstract class Shape serves as a **blueprint,** providing common methods like displayInfo(), while the subclasses implement the specific behavior for their area() methods.
- In this example, the abstract class Shape helps achieve abstraction and allows us to create a **hierarchy of shapes** with shared characteristics. It demonstrates how abstract classes are used to define common behavior while leaving specific implementations to their subclasses.
- In Java, the abstract keyword is used to **declare an abstract class or an abstract method**. It is a fundamental concept in object-oriented programming that allows you to define blueprints for classes and methods without providing a complete implementation.
- Abstract classes and methods are meant to be **subclassed and overridden by their concrete subclasses.**

INTERFACES IN JAVA:

- In Java, an interface is a **blueprint for a class**, similar to an abstract class, but with a different purpose and set of rules.
- An interface defines a set of abstract methods **(methods without implementation**) and constant fields **(implicitly public, static, and final**).
- Classes that implement an interface must provide concrete implementations for all the abstract methods declared in the interface.
- Interfaces allow you to achieve abstraction and define a contract that classes must adhere to, promoting code **flexibility and multiple inheritance** through interface implementation.

Key points about interfaces in Java:

- An interface is declared using the **interface keyword.**
- Interfaces can only have **abstract methods** and constant fields (**public, static, and final**).
- All methods declared in an interface are implicitly public and abstract. You don't need to use the public and abstract modifiers explicitly.
- Classes implement interfaces using the **implements keyword.**
- A class can implement **multiple interfaces,** allowing Java to achieve **multiple inheritance** through interface implementation.

Real-World Example: Vehicle Interface

Let's consider a real-world example of a Vehicle interface to illustrate the concept of interfaces:

Code:

```
// Interface: Vehicle
interface Vehicle {
    // Abstract method: start (to be implemented by classes)
    void start();

    // Abstract method: stop (to be implemented by classes)
    void stop();

    // Constant field: MAX_SPEED
    int MAX_SPEED = 100; // Public, Static, and Final
}
```

In this example, we have defined an interface Vehicle. It contains two abstract methods, start() and stop(), which must be implemented by classes that implement this interface. Additionally, it has a constant field MAX_SPEED, which is implicitly public, static, and final.

Now, let's create a class Car that implements the Vehicle interface:

Code:

```
// Class: Car (implements Vehicle)
class Car implements Vehicle {
    // Implementing the 'start' method for Car
    @Override
    public void start() {
        System.out.println("Car is starting.");
    }

    // Implementing the 'stop' method for Car
    @Override
    public void stop() {
        System.out.println("Car is stopping.");
    }
}
```

NOTE:

In this class, we have implemented both the start() and stop() methods defined in the Vehicle interface. The Car class is now considered to be a concrete implementation of the Vehicle interface.

Let's use the Car class in the Main class:

Code:

```
public class Main {
    public static void main(String[] args) {
        Vehicle myCar = new Car();

        myCar.start(); // Output: Car is starting.
        myCar.stop();  // Output: Car is stopping.

        System.out.println("Max Speed: " + Vehicle.MAX_SPEED);
// Output: Max Speed: 100
    }
}
```

- As you can see, we created an object of the Car class using the Vehicle interface reference. This demonstrates how Java achieves abstraction through interfaces. The Car class adheres to the contract defined by the Vehicle interface, providing **concrete implementations** for its abstract methods.
- In this example, the Vehicle interface acts as a blueprint that ensures any class implementing it has defined behavior for **starting and stopping, promoting code flexibility and modularity.**
- Interfaces are **powerful tools** in Java, allowing you to define contracts that multiple classes can fulfill, and they enable you to through interface implementation.

DIFFERENCE BETWEEN ABSTRACT AND INTERFACE IN JAVA:

In Java, both interfaces and abstract classes are used to achieve abstraction and provide blueprints for other classes, but they have different purposes, rules, and use cases.

Here are the key differences between interfaces and abstract classes:

Constructor:

- **Interface:** An interface cannot have a constructor because interfaces cannot be instantiated directly. They are not used to create objects.
- **Abstract Class:** An abstract class can have constructors. These constructors are invoked when a subclass is instantiated, allowing for common initialization logic.

Purpose:

- **Interface:** An interface defines a contract that a class must adhere to. It only contains method signatures (abstract methods) and constant fields (implicitly public, static, and final). It does not have any implementation of methods.
- **Abstract Class:** An abstract class is a class that cannot be instantiated on its own and serves as a blueprint for other classes. It can have both abstract methods (methods without a body) and concrete methods (methods with a body). Abstract classes can also have instance variables.

Multiple Inheritance:

- **Interface:** A class in Java can implement multiple interfaces, allowing for multiple inheritance of behavior. This is particularly useful when a class wants to conform to different contracts.
- **Abstract Class:** Java does not support multiple inheritance for classes. A class can extend only one abstract class, so it is limited to inheriting behavior from a single abstract class.

Method Implementation:

- **Interface:** All methods in an interface are implicitly public and abstract. Implementing classes must provide concrete implementations for all the abstract methods declared in the interface.
- **Abstract Class:** An abstract class can have both abstract methods (without implementation) and concrete methods (with implementation). Subclasses that extend the abstract class can choose to implement or override the abstract methods while inheriting the concrete methods.

Access Modifiers:

- **Interface:** All methods in an interface are implicitly public, and all fields are implicitly public, static, and final. You cannot use other access modifiers (e.g., private, protected) for methods or fields within an interface.
- **Abstract Class:** Abstract classes can have different access modifiers for their methods and fields, including public, protected, private, and default (package-private).

Use cases:

- Use interfaces when you want to define a contract that multiple unrelated classes can fulfill, promoting code flexibility and multiple inheritance.
- Use abstract classes when you want to define a common base class with some shared behavior and leave certain methods to be implemented by its subclasses.

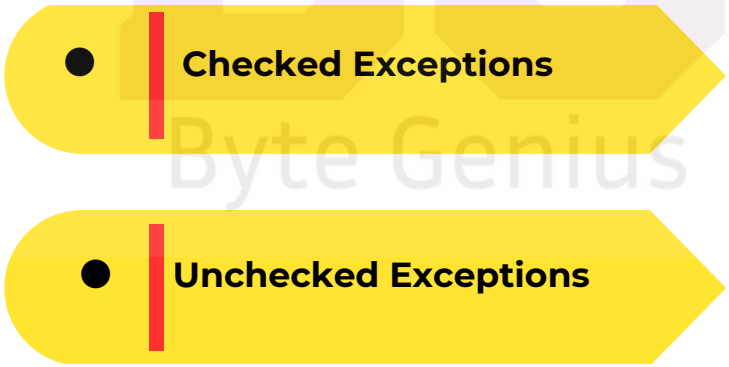
In summary, interfaces are used to define contracts, while abstract classes are used to provide common behavior and **partial implementation.**

- Interfaces allow multiple inheritance through **implementation**, while abstract classes provide a single inheritance hierarchy.
- The choice between using an **interface or an abstract class** depends on the design and requirements of your application.

EXCEPTIONS IN JAVA :

- In Java, an **exception is an event** that occurs during the execution of a program, disrupting the **normal flow of code.**
- When an exceptional situation arises, Java throws an exception, and the program **can catch and handle the exception** to recover or gracefully exit instead of terminating abnormally.
- Exceptions are **used to handle errors, unexpected events, or exceptional conditions in a controlled manner.**
- Java provides a **built-in mechanism** to deal with exceptions through the use of **try-catch blocks.**
- **A try** block contains the code that might raise an **exception,** and a **catch block catches and handles the exception.**
- This way, the program can continue executing without **crashing,** and the developer can address the **exceptional situation appropriately.**

Types of Exceptions in Java:



Checked exceptions:

- Checked exceptions are the exceptions that must be **handled explicitly by the programmer.**
- They are checked at **compile-time**, and the compiler forces you to handle them using a **try-catch block or declare** them in the method signature using the throws keyword.
- Checked exceptions are typically used for conditions that can be anticipated and recovered from, **such as file I/O errors, network issues, etc.**

Example:

Code:

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

public class FileExample {
    public static void main(String[] args) {
        try {
            BufferedReader reader = new BufferedReader(new
FileReader("file.txt"));
            String line = reader.readLine();
            System.out.println("First line: " + line);
            reader.close();
        } catch (IOException e) {
            System.err.println("Error reading the file: " + e.getMessage()
        }
    }
}
```

Unchecked exceptions (Runtime Exceptions):

- Unchecked exceptions, also known as **runtime exceptions,** do not need to be **explicitly handled**.
- They are not checked at compile-time, and Java does not enforce **catching** or declaring them with the **throws** keyword.
- Unchecked exceptions are typically caused by programming errors such as **dividing by zero, accessing an invalid index in an array,** etc.

Example:

Code:

```
public class DivisionExample {
    public static void main(String[] args) {
        int numerator = 10;
        int denominator = 0;
        try {
            int result = numerator / denominator;
            System.out.println("Result: " + result);
        } catch (ArithmeticException e) {
            System.err.println("Error: " + e.getMessage());
        }
    }
}
```

- In both examples, we use **try-catch blocks** to handle potential exceptions. In the first example, a checked exception **(IOException)** is handled when reading a file, while in the second example, an unchecked exception (**ArithmeticException**) is handled when dividing by zero.
- **Handling exceptions** allows the program to handle errors gracefully, provide useful feedback to users, and prevent the program from terminating unexpectedly. Proper exception handling is an essential aspect of **writing reliable and robust Java applications.**

MULTITHREADING IN JAVA :

- Multithreading in Java allows a program to execute **multiple threads concurrently,** enabling it to perform multiple tasks at the same time. **A thread is a lightweight sub-process** within a program that can execute independently.
- Multithreading can improve the **performance** and **responsiveness** of a Java application, especially when dealing with tasks that can be executed in **parallel** or when handling **time-consuming** operations without blocking the main execution.

To work with multithreading in Java, you need to use the **Thread class** or **implement the Runnable interface:**

Real-World Example: Printing Numbers Using Threads

Let's create a simple example that demonstrates multithreading by printing numbers using two threads:

Code:

```
class NumberPrinter extends Thread {
    private int start;
    private int end;

    public NumberPrinter(int start, int end) {
        this.start = start;
        this.end = end;
    }
```

Contd:

Code:

```
    @Override
    public void run() {
        for (int i = start; i <= end; i++) {
            System.out.println(Thread.currentThread().getName() + ": " + i);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        int totalNumbers = 10;
        int mid = totalNumbers / 2;

        // Create two threads
        NumberPrinter thread1 = new NumberPrinter(1, mid);
        NumberPrinter thread2 = new NumberPrinter(mid + 1, totalNumbers);

        // Start both threads
        thread1.start();
        thread2.start();

        // Wait for both threads to finish
        try {
            thread1.join();
            thread2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Main thread exiting.");
    }
}
```

Explanation:

1. We create a NumberPrinter class that extends the Thread class and overrides the **run() method.** This method contains the task we want the **thread to perform**, which is printing numbers from start to end.
2. In the **Main class,** we create **two instances** of the NumberPrinter class, each responsible for printing a different range of numbers.
3. We start the threads using the s**tart() method**. This method causes the run() method of the NumberPrinter class to be executed concurrently.
4. We use the **join()** method to ensure that the main thread waits for both thread1 and thread2 to finish before it prints **"Main thread exiting."** This helps in synchronizing the threads and ensures that the program exits gracefully.

When you run the above code, you will see that both threads are executing concurrently and printing numbers in their respective ranges. The order of execution may vary from run to run as it depends on the scheduler and the execution environment.

- **Multithreading** is useful when dealing with tasks that can be performed independently and concurrently, such as handling user requests, **parallel processing,** and **optimizing resource utilization.**
- However, it also adds complexity to the code, so it's essential to **design and implement multithreaded applications carefully to ensure correctness and avoid potential issues.**

Important Note:
That multithreading requires careful handling of shared resources to avoid issues like race conditions and deadlocks. Advanced techniques and tools like synchronization, locks, and thread pools are used to manage concurrency and thread safety in more complex multithreaded applications.