**Class and object**

**Class:**

**A class in Java is like a blueprint or template for creating objects. It defines properties (attributes) and behaviors (methods) that the objects created from the class will have.**

**Example: Think of a class like the blueprint for a table. It specifies the dimensions, material, and design. However, the blueprint itself is not a table; it's just a plan for creating tables.**

**In programming terms, you are the class that gives the blueprint (instructions) to a carpenter (the program). The carpenter follows these instructions to create a real table (object).**

**Object:**

**An object is an instance of a class. It is a real-world entity created based on the blueprint (class). In Java, everything revolves around objects. Once the JVM creates an object using the class as a blueprint, the object will have the properties and behaviors defined by the class.**

**Example: If the blueprint is the class, then the actual table made by the carpenter is the object. Each table (object) may have its own state (specific size, color, etc.) but follows the general structure defined by the blueprint (class).**

**Stack And Heap Memory**

**1. Stack Memory:**

* **Stack memory is used for static memory allocation and storing local variables.**
* **It holds primitive data types (int, float, char, etc.) and references to objects (but not the objects themselves).**
* **Each time a method is called, a new block (or frame) is created on the stack for that method. This block contains all the method's local variables.**
* **When the method execution completes, its block is removed from the stack (last in, first out - LIFO structure).**

**Example of Stack Memory:**

**public class StackExample {**

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

**int a = 10; // stored in Stack (primitive)**

**int b = 20; // stored in Stack (primitive)**

**add(a, b); // method call creates a new stack frame for the method 'add'**

**}**

**public static void add(int x, int y) {**

**int sum = x + y; // x, y, and sum are stored in the stack frame for 'add'**

**System.out.println(sum);**

**}**

**}**

**Explanation:**

* **Variables a and b are stored in the main() method's stack frame.**
* **When the add() method is called, x, y, and sum are stored in the add() method's stack frame.**
* **Once the add() method finishes execution, its stack frame is destroyed, and x, y, and sum are removed.**

**2. Heap Memory:**

* **Heap memory is used for dynamic memory allocation, and it stores all objects created in Java using the new keyword.**
* **It holds the actual objects, and the references to these objects are stored in the stack.**
* **Heap memory is managed by the Java Garbage Collector, which reclaims memory used by objects that are no longer referenced.**

**Example of Heap Memory:**

**public class HeapExample {**

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

**// Person object is created in the heap, reference stored in the stack**

**Person person1 = new Person("John", 25);**

**Person person2 = new Person("Alice", 30);**

**}**

**}**

**class Person {**

**String name; // stored in heap**

**int age; // stored in heap**

**Person(String name, int age) {**

**this.name = name;**

**this.age = age;**

**}**

**}**

**Explanation:**

* **The two Person objects (person1 and person2) are created in the heap memory.**
* **However, their references (or pointers) are stored in the stack memory in the main() method's stack frame.**
* **The name and age fields of each object are also stored in the heap memory.**

**Static Variable In Java**

**A static variable in Java (also known as a class variable) is a variable that is shared across all instances of a class. Unlike instance variables that are unique to each object, a static variable is created once and shared by all objects of that class. Let's explore static variables in detail:**

**1. Characteristics of a Static Variable:**

* **Belongs to the class, not objects: A static variable is associated with the class itself rather than any specific instance (object) of the class. This means it is shared among all instances of the class.**
* **Single copy across all instances: Only one copy of the static variable exists, no matter how many objects of the class are created. All objects refer to the same copy of the static variable.**
* **Memory allocation: Static variables are stored in the method area (or the class area in the JVM) and are allocated memory only once, when the class is loaded.**
* **Access via class name: Static variables can be accessed directly using the class name, without needing to create an object of the class. However, they can also be accessed via objects.**

**2. Declaration of Static Variable:**

**Static variables are declared using the static keyword inside a class but outside any method, constructor, or block.**

**class MyClass {**

**// Static variable**

**static int count = 0;**

**}**

**3. Use Cases of Static Variables:**

* **To share common data: Since static variables are shared among all instances, they are often used for storing common data that needs to be accessed or modified by all instances. For example, a variable that tracks the number of objects created can be declared as static.**
* **Memory efficiency: Static variables are used when a variable needs to be shared but creating multiple copies of it would be inefficient. For example, a static variable can be used to hold configuration settings for an application.**

**4. How Static Variables Work:**

* **Without Objects: A static variable can be accessed directly using the class name, even without creating an object of the class.**

**MyClass.count = 5;**

* **With Objects: Static variables can also be accessed through objects, but it is not common practice.**

**MyClass obj1 = new MyClass();**

**MyClass obj2 = new MyClass();**

**// Accessing static variable using objects**

**obj1.count = 10;**

**obj2.count = 20;**

**// The value of 'count' will be 20 for both objects**

**System.out.println(obj1.count); // Output: 20**

**System.out.println(obj2.count); // Output: 20**

**5. Example: Counting the Number of Objects Created**

**Let’s look at an example where we use a static variable to keep track of the number of objects created for a class.**

**class MyClass {**

**// Static variable to count number of objects**

**static int objectCount = 0;**

**// Constructor**

**MyClass() {**

**// Incrementing the count every time an object is created**

**objectCount++;**

**}**

**// Method to display the object count**

**static void displayObjectCount() {**

**System.out.println("Number of objects created: " + objectCount);**

**}**

**}**

**public class Main {**

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

**// Creating objects**

**MyClass obj1 = new MyClass();**

**MyClass obj2 = new MyClass();**

**MyClass obj3 = new MyClass();**

**// Displaying the count of objects created**

**MyClass.displayObjectCount(); // Output: "Number of objects created: 3"**

**}**

**}**

**In this example:**

* **The static variable objectCount keeps track of the number of objects created.**
* **Since objectCount is static, all instances of MyClass share the same variable.**
* **Every time a new object is created, the constructor increments objectCount, and the total number of objects created can be retrieved using the displayObjectCount method.**

**Static Method in Java**

**A static method in Java is a method that belongs to the class rather than instances (objects) of the class. Like static variables, static methods can be called without creating an object of the class. They are commonly used to perform operations that are related to the class itself or to operate on static variables.**

**Key Characteristics of Static Methods:**

* **Belongs to the class: A static method is associated with the class, not any particular instance. It is defined using the static keyword.**
* **Access via class name: Static methods can be called directly using the class name, without the need to instantiate an object.**
* **No access to instance variables: Static methods cannot access instance variables or instance methods directly, because they are not tied to a specific object.**
* **Can access static variables and static methods: Static methods can only access static variables and other static methods directly.**
* **Cannot use this or super keywords: Since static methods are not associated with any particular instance, they cannot use this (which refers to the current object) or super (which refers to the parent class).**

**Declaration of a Static Method:**

**Static methods are declared using the static keyword.**

**class MyClass {**

**static void myStaticMethod() {**

**// Method code**

**}**

**}**

**Calling a Static Method:**

**Static methods can be called:**

1. **Using the class name:**

**MyClass.myStaticMethod();**

1. **Using an object (although not recommended):**

**MyClass obj = new MyClass();**

**obj.myStaticMethod(); // This works, but it's not the best practice.**

**Example: Static Method in Action**

**Let’s see an example where a static method is used to perform an operation related to the class, such as counting the number of objects created.**

**class MyClass {**

**// Static variable to keep track of object count**

**static int count = 0;**

**// Constructor**

**MyClass() {**

**count++; // Increment count when an object is created**

**}**

**// Static method to display the object count**

**static void displayCount() {**

**System.out.println("Total objects created: " + count);**

**}**

**}**

**public class Main {**

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

**MyClass obj1 = new MyClass();**

**MyClass obj2 = new MyClass();**

**// Calling the static method using the class name**

**MyClass.displayCount(); // Output: "Total objects created: 2"**

**}**

**}**

**Static Block in Java (Static Initialization Block)**

**A static block (also called a static initialization block) in Java is a block of code that is executed only once when the class is loaded into memory by the Java ClassLoader. It is mainly used to initialize static variables or execute logic that needs to run before the object creation or any static method is called.**

**Key Points about Static Blocks:**

* **Executed when the class is loaded: The static block is executed as soon as the class is loaded into memory, even before the main method or any constructor is executed.**
* **Runs only once: It runs only one time per class, no matter how many objects are created.**
* **Multiple static blocks: You can have multiple static blocks in a class, and they are executed in the order they appear.**
* **Used for static variable initialization: Static blocks are often used to initialize static variables or perform any setup that's required before the class is used.**

**Syntax of a Static Block:**

**class MyClass {**

**static {**

**// Static block code**

**}**

**}**

**Example: Static Block in Action**

**class MyClass {**

**static int a;**

**static int b;**

**// Static block to initialize static variables**

**static {**

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

**a = 10;**

**b = 20;**

**}**

**// Constructor**

**MyClass() {**

**System.out.println("Constructor executed");**

**}**

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

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

**// Accessing static variables**

**System.out.println("a = " + a); // a = 10**

**System.out.println("b = " + b); // b = 20**

**// Creating an object of MyClass**

**MyClass obj = new MyClass();**

**}**

**}**

**Encapsulation in Java**

**Imagine a capsule (like a pill). The medicine inside the capsule is protected, and you can’t see or directly access it. You just take the capsule, and it works. Similarly, in Java, encapsulation hides the internal data of an object and only allows access through defined methods.**

**How Encapsulation Works:**

1. **Private Variables: You make the variables (data) of a class private so that no one outside the class can directly access or change them.**
2. **Public Methods: You provide public methods (getters and setters) to access and modify the private variables. These methods control how the data is accessed and changed.**

**Example:**

**class Person {**

**// Private variable, can't be accessed directly outside this class**

**private String name;**

**// Getter method to access the private variable**

**public String getName() {**

**return name;**

**}**

**// Setter method to modify the private variable**

**public void setName(String newName) {**

**name = newName;**

**}**

**}**

**public class Main {**

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

**Person person = new Person();**

**// Using setter to set the name**

**person.setName("Onkar");**

**// Using getter to get the name**

**System.out.println(person.getName()); // Output: Onkar**

**}**

**}**

**Constructor in Java**

**A constructor is a special type of method in Java that is used to initialize objects. It is called when an instance of a class is created. Constructors are similar to methods but differ in that they:**

1. **Have the same name as the class.**
2. **Do not return any type, not even void.**
3. **Are automatically called when an object is created.**

**Types of Constructors**

**There are two main types of constructors in Java:**

1. **Default Constructor (No-argument constructor)**
2. **Parameterized Constructor**

**1. Default Constructor**

* **A default constructor is provided by the Java compiler if no constructors are explicitly defined in the class.**
* **It initializes object with default values (e.g., 0 for integers, null for objects).**

**Example:**

**class Car {**

**String model;**

**int year;**

**// Default constructor**

**Car() {**

**model = "Unknown";**

**year = 2024;**

**}**

**}**

**Usage:**

**Car myCar = new Car();**

**2. Parameterized Constructor**

* **A parameterized constructor allows passing values to initialize an object with specific data.**

**Example:**

**class Car {**

**String model;**

**int year;**

**// Parameterized constructor**

**Car(String m, int y) {**

**model = m;**

**year = y;**

**}**

**}**

**Usage:**

**Car myCar = new Car("Tesla", 2023);**

**Anonymous Object**

Anonymous Object Means where we just create a object not store in any Variable, we can not reuse it.

**class Car {**

**Car() {**

**s.o.println(“Object Created”);**

**}**

**Public void show(){**

**s.o.println(“Inside the Method”);**

**}**

**}**

In main method

Main(){

new Car();

}

**super() and this() method**

**1. super()**

* **Purpose**: The super() method is used to invoke the constructor of the **parent (superclass)**. It must be the first statement in a subclass constructor.
* **Why use super()**: It allows the subclass to initialize the fields or behavior inherited from the parent class. If super() is not explicitly called, the compiler automatically inserts a call to the no-argument constructor of the parent class.

**Key Points:**

* Used to call the parent class's constructor.
* If you don’t specify super(), Java implicitly adds a call to the no-argument constructor of the superclass.
* You can also pass arguments to super() to invoke a specific constructor of the parent class.

**Example:**

class A {

public A() {

System.out.println("in A's default constructor");

}

public A(int n) {

System.out.println("in A's parameterized constructor");

}

}

class B extends A {

public B() {

super(); // Calls A's default constructor

System.out.println("in B's default constructor");

}

public B(int n) {

super(n); // Calls A's parameterized constructor

System.out.println("in B's parameterized constructor");

}

}

public class Main {

public static void main(String[] args) {

B obj1 = new B(); // Calls A's default, then B's default

B obj2 = new B(5); // Calls A's parameterized, then B's parameterized

}

}

Output :-

in A's default constructor

in B's default constructor

in A's parameterized constructor

in B's parameterized constructor

**2. this()**

* **Purpose**: The this() method is used to call the **current class's** constructor. Like super(), it must be the first statement in the constructor.
* **Why use this()**: It allows constructor chaining within the same class. You can avoid repeating code by using this() to call one constructor from another in the same class, passing appropriate arguments.

**Key Points:**

* Used to call another constructor in the same class.
* Helps in reducing redundant code by chaining constructors.
* If this() is used, it must be the first statement in the constructor.

class A {

public A() {

this(10); // Calls the parameterized constructor

System.out.println("in A's default constructor");

}

public A(int n) {

System.out.println("in A's parameterized constructor with value: " + n);

}

}

public class Main {

public static void main(String[] args) {

A obj = new A(); // Calls A's default constructor, which then calls the parameterized constructor

}

}

Output ;-

in A's parameterized constructor with value: 10

in A's default constructor

**Method Overriding in Java**

**Method overriding** is a feature in Java that allows a subclass (child class) to provide a specific implementation for a method that is already defined in its superclass (parent class). When a subclass has a method with the **same name, return type, and parameters** as a method in its superclass, the method in the subclass overrides the method in the superclass.

**Key Points of Method Overriding:**

1. **Same method signature**: The overriding method must have the same name, return type, and parameters as the method in the parent class.
2. **Inheritance**: Method overriding requires a superclass and a subclass. The method that gets overridden must be defined in the parent class.
3. **Runtime Polymorphism**: Method overriding supports runtime polymorphism, meaning the decision to call the overridden method is made at runtime based on the object type.
4. **Access modifier**: The overriding method cannot reduce the visibility of the method it overrides. For example, if the parent method is public, the overriding method cannot be private or protected.
5. **super keyword**: The super keyword can be used in the subclass to refer to the parent class's version of the method.
6. **Static methods and constructors** cannot be overridden. Overriding only applies to instance methods.

**Example of Method Overriding:**

Let’s see how overriding works with an example.

**Superclass (Parent Class):**

class Animal {

// Method to be overridden

public void sound() {

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

}

}

**Subclass (Child Class):**

class Dog extends Animal {

// Overriding the sound() method

@Override

public void sound() {

System.out.println("Dog barks");

}

}

**Test the Method Overriding:**

public class Main {

public static void main(String[] args) {

Animal animal = new Animal(); // Object of parent class

animal.sound(); // Calls Animal's version of sound()

Dog dog = new Dog(); // Object of child class

dog.sound(); // Calls Dog's version of sound() (overridden method)

Animal animalDog = new Dog(); // Parent reference, child object

animalDog.sound(); // Calls Dog's version of sound() (runtime polymorphism)

}

}

**Output:**

Animal makes a sound

Dog barks

Dog barks

Packages in Java

OnlineShoppingSystem/

└── src/

├── com/

│ └── onlineshop/

│ ├── products/

│ │ ├── Product.java

│ │ └── Electronics.java

│ ├── users/

│ │ ├── User.java

│ │ └── Customer.java

│ ├── orders/

│ │ ├── Order.java

│ │ └── OrderManager.java

│ └── main/

│ └── ShoppingApp.java

└── README.md

**Packages In Java**

1. **Public**:

* **Access Level**: Everywhere.
* **Description**: If a class, method, or variable is declared as public, it can be accessed from any other class, regardless of whether it's in the same package or not.
* **Example**:

public class MyClass {

public int myVar;

public void myMethod() {

System.out.println("Public method");

}

}

2. **Private**:

* **Access Level**: Only within the same class.
* **Description**: If something is declared private, it can only be accessed within the class where it's defined. This is useful for encapsulating data.
* **Example**:

public class MyClass {

private int myVar;

private void myMethod() {

System.out.println("Private method");

}

}

3 **Protected**:

* **Access Level**: Same package and subclasses.
* **Description**: A protected member can be accessed within the same package and also in subclasses (even if the subclass is in a different package).
* **Example**:

public class MyClass {

protected int myVar;

protected void myMethod() {

System.out.println("Protected method");

}

}

4 **Default (Package-Private)**:

* **Access Level**: Only within the same package.
* **Description**: When no access modifier is specified, it defaults to package-private, meaning it can only be accessed by other classes in the same package. This is also called **default** access.
* **Example**:

class MyClass {

int myVar; // default access

void myMethod() {

System.out.println("Default method");

}

}

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **public** | Yes | Yes | Yes | Yes |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **protected** | Yes | Yes | Yes | No |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **default** | Yes | Yes | No | No |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **private** | Yes | No | No | No |

**Abstract Class and Abstract Method**

**1 Abstract Method: An abstract method is a method that is declared without an implementation (i.e., without a body). It is used when you have an idea of what the method should do, but the specific implementation will be provided by a subclass. You declare an abstract method using the abstract keyword.**

**2 Abstract Class: An abstract class is a class that cannot be instantiated directly. It may contain both abstract methods (without implementation) and non-abstract methods (with implementation). To create an object of an abstract class, it must be extended by a subclass.**

**Key Points:**

1. **An abstract method must be declared in an abstract class.**
2. **If a class contains even one abstract method, it must be declared as an abstract class.**
3. **A subclass that extends an abstract class is required to implement all the abstract methods from the parent class. If it does not, the subclass must also be declared as abstract.**
4. **We can not create Object of Abstract class But we can take refrence of Abstract class**

**// Abstract class**

**abstract class Animal {**

**// Abstract method (no implementation)**

**public abstract void makeSound();**

**// Non-abstract method (with implementation)**

**public void sleep() {**

**System.out.println("Sleeping...");**

**}**

**}**

**// Subclass extending the abstract class**

**class Dog extends Animal {**

**// Providing implementation for the abstract method**

**public void makeSound() {**

**System.out.println("Bark");**

**}**

**}**

**public class Main {**

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

**Dog myDog = new Dog(); // Creating object of the subclass**

**myDog.makeSound(); // Output: Bark**

**myDog.sleep(); // Output: Sleeping...**

**}**

**}**

**Inner Classes in Java**

In Java, an inner class is a class that is defined inside the body of another class. Inner classes can be useful for logically grouping classes together that are only used in one place. Moreover, they allow access to the members (including private members) of the outer class, providing tight coupling between the two classes.

**Types of Inner Classes:**

1. **Non-static (Regular) Inner Class**
2. **Static Nested Class**
3. **Local Inner Class**
4. **Anonymous Inner Class**

**1. Non-static (Regular) Inner Class**

A regular inner class is associated with an instance of the outer class. It cannot define any static members, and to instantiate it, you need an instance of the outer class.

**Key Points:**

* Can access all members of the outer class, including private fields.
* Requires an instance of the outer class to instantiate it.

class Outer {

private String message = "Hello from Outer Class";

class Inner {

void displayMessage() {

System.out.println(message); // Can access private members of Outer class

}

}

public static void main(String[] args) {

Outer outerObj = new Outer();

Outer.Inner innerObj = outerObj.new Inner(); // Create inner class object

innerObj.displayMessage(); // Output: Hello from Outer Class

}

}

**2. Static Nested Class**

A static nested class is treated like a static member of the outer class. It does not require an instance of the outer class to be instantiated and can only access static members of the outer class.

**Key Points:**

* Does **not** require an instance of the outer class to create an object.
* Can only access static members of the outer class.

class Outer {

static String staticMessage = "Static message from Outer Class";

static class Nested {

void display() {

System.out.println(staticMessage); // Can access only static members of Outer class

}

}

public static void main(String[] args) {

Outer.Nested nestedObj = new Outer.Nested(); // No need for Outer class instance

nestedObj.display(); // Output: Static message from Outer Class

}

}

**3. Local Inner Class**

A local inner class is defined within a method of an outer class. It is only accessible within the method in which it is defined. A local inner class can access local variables and parameters of the method, but only if they are declared as final or are effectively final.

**Key Points:**

* Defined inside a method or block.
* Can access final or effectively final variables of the method in which it is defined.

class Outer {

void outerMethod() {

final String localMessage = "Local Inner Class Message";

class LocalInner {

void display() {

System.out.println(localMessage); // Can access local variables marked final

}

}

LocalInner localInner = new LocalInner();

localInner.display(); // Output: Local Inner Class Message

}

public static void main(String[] args) {

Outer outerObj = new Outer();

outerObj.outerMethod();

}

}

**interface in Java**

An **interface** in Java is a reference type, similar to a class, that can contain only constants, method signatures (abstract methods), default methods, static methods, and nested types. Interfaces cannot have constructors, instance fields, or methods with implementations (except for default and static methods, as introduced in Java 8).

Interfaces are used to define a contract, or a set of methods, that a class must implement. It is a way to achieve **full abstraction** in Java because all methods in an interface are abstract by default (unless they are default or static).

**Key Features of Interfaces:**

1. **Abstract methods by default**: Every method in an interface is **public and abstract** by default, meaning you only declare the method signature in the interface, and the implementing class provides the actual method body.
2. **No method implementation (except for default and static methods)**: Interfaces do not contain implementations for their methods (except for Java 8+ default and static methods).
3. **Multiple Inheritance**: A class can implement multiple interfaces, allowing a form of multiple inheritance (which is not possible with classes in Java).
4. **Fields are public, static, and final**: Any fields declared in an interface are implicitly public, static, and final (constant).
5. **Cannot be instantiated**: Interfaces, like abstract classes, cannot be instantiated directly.

// Interface declaration

interface Animal {

void eat(); // Abstract method

void sleep(); // Abstract method

}

// Class implementing the interface

class Dog implements Animal {

public void eat() {

System.out.println("Dog is eating.");

}

public void sleep() {

System.out.println("Dog is sleeping.");

}

}

public class Main {

public static void main(String[] args) {

Dog myDog = new Dog();

myDog.eat(); // Output: Dog is eating.

myDog.sleep(); // Output: Dog is sleeping.

}

}

**1. Multiple Inheritance Using Interfaces**

Java does not support multiple inheritance with classes (i.e., a class cannot extend more than one class). However, a class **can implement multiple interfaces**, allowing a form of multiple inheritance.

interface Flyable {

void fly();

}

interface Swimmable {

void swim();v

}

class Duck implements Flyable, Swimmable {

public void fly() {

System.out.println("Duck is flying.");

}

public void swim() {

System.out.println("Duck is swimming.");

}

}

public class Main {

public static void main(String[] args) {

Duck duck = new Duck();

duck.fly(); // Output: Duck is flying.

duck.swim(); // Output: Duck is swimming.

}

**Key Points :-**

* 1. **We can add Inheritance in Interface .**
  2. **Class – Class 🡪 extends**
  3. **Class – Interface 🡪 implements**
  4. **Interface – interface 🡪 extends**

**Enums in Java**

**Enum in Java - Simple Example**

Enums are used to define a set of constant values. Let's break it down with an easy example, like defining the **days of the week**.

**1. Basic Enum Example: Days of the Week**

Imagine you want to represent the days of the week in your program. Instead of using strings, you can use an enum to ensure that only valid days can be assigned to a variable.

// Defining an Enum

public enum Day {

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

}

public class EnumExample {

public static void main(String[] args) {

Day today = Day.MONDAY; // Assigning a value from the enum

// Checking the value of today using a switch statement

switch (today) {

case MONDAY:

System.out.println("Today is Monday. Time to start the week!");

break;

case FRIDAY:

System.out.println("It's Friday! Weekend is near!");

break;

case SUNDAY:

System.out.println("It's Sunday, time to relax!");

break;

default:

System.out.println("It's just another day of the week."); }}}

**3. Enum with Methods Example: Traffic Lights**

Let's make it a bit more interesting by adding **actions** to an enum. Consider traffic lights with different actions (Stop, Go, Slow down).

// Enum for Traffic Lights

public enum TrafficLight {

RED("Stop"),

YELLOW("Slow down"),

GREEN("Go");

private String action;

// Constructor to set the action for each light

private TrafficLight(String action) {

this.action = action;

}

// Method to get the action

public String getAction() {

return this.action;

}

}

public class TrafficLightExample {

public static void main(String[] args) {

TrafficLight light = TrafficLight.RED; // Assigning RED light

// Printing the action associated with the traffic light

System.out.println("The light is " + light + ": " + light.getAction());

}

}

**Annotations**

Annotations in Java are a form of metadata that provide data about a program, but they do not change its actual code execution. Annotations are used to provide information to the compiler, to generate code, or to help the developer write cleaner code.

**1. Built-in Annotations**

These are annotations provided by the Java language itself. They are frequently used in day-to-day programming.

**1.1 @Override**

* **Purpose**: Indicates that a method is overriding a method from a superclass.
* **Usage**: Helps to avoid errors, ensuring that a method is correctly overriding the parent class method.

class Parent {

public void display() {

System.out.println("Parent display");

}

}

class Child extends Parent {

@Override

public void display() {

System.out.println("Child display");

}

}

**1.2 @Deprecated**

* **Purpose**: Marks a method, class, or field as deprecated, meaning it is no longer recommended to be used and might be removed in future versions.
* **Usage**: Signals to developers that a piece of code is outdated.

@Deprecated

public void oldMethod() {

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

}

**1.3 @SuppressWarnings**

* **Purpose**: Instructs the compiler to suppress specific warnings.
* **Usage**: Commonly used when we want to hide warnings about unchecked operations, unused variables, etc.

@SuppressWarnings("unchecked")

public void someMethod() {

List list = new ArrayList(); // No warning will be shown here

list.add("Hello");

}

**1.4 @SafeVarargs**

* **Purpose**: Suppresses warnings for heap pollution when using varargs (variable-length argument lists) with generics.
* **Usage**: Commonly used in methods with varargs and generics.

@SafeVarargs

public final <T> void safeMethod(T... elements) {

for (T element : elements) {

System.out.println(element);

}

}

**2. Meta-Annotations**

Meta-annotations are annotations that apply to other annotations. Java provides several meta-annotations:

**2.1 @Retention**

* **Purpose**: Specifies how long annotations are to be retained (kept). There are three retention policies:
  + SOURCE: Annotations are discarded by the compiler.
  + CLASS: Annotations are kept in the class file but ignored by the JVM.
  + RUNTIME: Annotations are kept in the class file and retained by the JVM during runtime.

@Retention(RetentionPolicy.RUNTIME)

public @interface MyAnnotation {

}

**2.2 @Target**

* **Purpose**: Restricts where the annotation can be applied (e.g., on a method, field, class).

@Target(ElementType.METHOD)

public @interface MethodAnnotation {

}

**Types of Interfaces**

**1. Normal Interface**

A **normal interface** is a typical interface in Java that can contain:

* Abstract methods (methods without a body)
* Default methods (methods with a body)
* Static methods

A class that implements a normal interface must provide implementations for all the abstract methods defined in that interface.

interface Animal {

void eat(); // abstract method

void sleep(); // abstract method

}

class Dog implements Animal {

@Override

public void eat() {

System.out.println("Dog is eating.");

}

@Override

public void sleep() {

System.out.println("Dog is sleeping.");

}

}

**2. Functional Interface**

A **functional interface** is an interface that contains only one abstract method. This is used mainly with **lambda expressions** and **method references** in Java. A functional interface can have any number of default and static methods, but it can only have one abstract method.

@FunctionalInterface

interface Greeting {

void sayHello(); // Only one abstract method

}

// Using a lambda expression to implement the functional interface

public class Main {

public static void main(String[] args) {

Greeting greeting = () -> System.out.println("Hello!");

greeting.sayHello();

}

}

**3. Marker Interface**

A **marker interface** is an interface that doesn't contain any methods or fields. It is used to mark or tag a class with special behavior or metadata. In essence, marker interfaces provide a way for the Java runtime or external tools to identify classes that implement them and treat them differently.

Some well-known marker interfaces in Java include:

* Serializable: Marks a class as capable of being serialized.
* Cloneable: Marks a class as capable of being cloned.

interface MarkerInterface {

// No methods or fields, just a marker

}

class MyClass implements MarkerInterface {

// This class is marked with MarkerInterface

}

public class Main {

public static void main(String[] args) {

MyClass obj = new MyClass();

if (obj instanceof MarkerInterface) {

System.out.println("MyClass is marked with MarkerInterface");

}

}

}

**Lambda Expression**

**What is a Lambda Expression?**

A **lambda expression** is essentially a shorter way to write anonymous classes or methods that can be passed around as arguments. It helps make the code more readable and concise, especially when dealing with functional programming.

In Java, lambda expressions are primarily used to implement **functional interfaces**. As you learned earlier, a functional interface is an interface with only **one abstract method**.

**Why use Lambda Expressions?**

* They make your code shorter and cleaner.
* They allow you to pass behavior as parameters.
* They help make Java more functional, bringing it closer to languages like Python, Scala, or JavaScript.

(parameter list) -> { body of the method }

Example 1: Without Lambda Expression (Traditional Approach)

interface Greeting {

void sayHello();

}

public class Main {

public static void main(String[] args) {

// Anonymous class implementation

Greeting greeting = new Greeting() {

@Override

public void sayHello() {

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

}

};

greeting.sayHello(); // Output: Hello, world!

}

}

Example 2: With Lambda Expression

interface Greeting {

void sayHello();

}

public class Main {

public static void main(String[] args) {

// Lambda expression implementation

Greeting greeting = () -> System.out.println("Hello, world!");

greeting.sayHello(); // Output: Hello, world!

}

}

**Custom Sorting in java**

**IMP**

import java.util.Arrays;

public class Main {

public static void main(String[] args) {

// Sample 2D array (multiple rows, 2 columns)

int[][] array = {

{3, 2},

{1, 4},

{5, 1},

{2, 3}

};

// Printing original array

System.out.println("Original array:");

printArray(array);

// Sorting the array by the second column (index 1)

Arrays.sort(array, (a, b) -> Integer.compare(a[1], b[1]));

// Printing the sorted array

System.out.println("\nSorted array (by second column):");

printArray(array);

}

// Helper function to print the 2D array

public static void printArray(int[][] array) {

for (int[] row : array) {

System.out.println(Arrays.toString(row));

}

}

}

**Errors in java**

**1. Compile-time Errors**

**Compile-time errors** occur when the code does not adhere to the rules of the Java language. These errors are detected by the **Java compiler** before the program runs. You cannot execute the program until these errors are fixed.

**Common Causes:**

* Syntax errors (missing semicolons, incorrect use of keywords, etc.)
* Missing classes or packages
* Incorrect data types

public class Main {

public static void main(String[] args) {

int number = "Hello"; // ERROR: Incompatible types (String assigned to int)

System.out.println(number);

}

}

**2. Runtime Errors**

**Runtime errors** occur when the program compiles successfully but encounters an error during execution. These errors usually occur due to invalid operations like dividing by zero, accessing an array out of bounds, or dereferencing a null pointer. They cause the program to terminate abnormally.

**Common Causes:**

* Division by zero
* Array index out of bounds
* Null pointer dereferencing

public class Main {

public static void main(String[] args) {

int a = 10;

int b = 0;

int result = a / b; // ERROR: Division by zero

System.out.println(result);

}}

**3. Logical Errors**

**Logical errors** are the most difficult to detect because the program compiles and runs without throwing any exceptions. However, the output or behavior of the program is not what you expected due to an error in the logic of the program.

**Common Causes:**

* Incorrect algorithms
* Mistakes in calculations
* Wrong conditional statements

public class Main {

public static void main(String[] args) {

int number = 5;

if (number > 10) {

System.out.println("Number is greater than 10");

} else {

System.out.println("Number is less than or equal to 10");

}

}

}

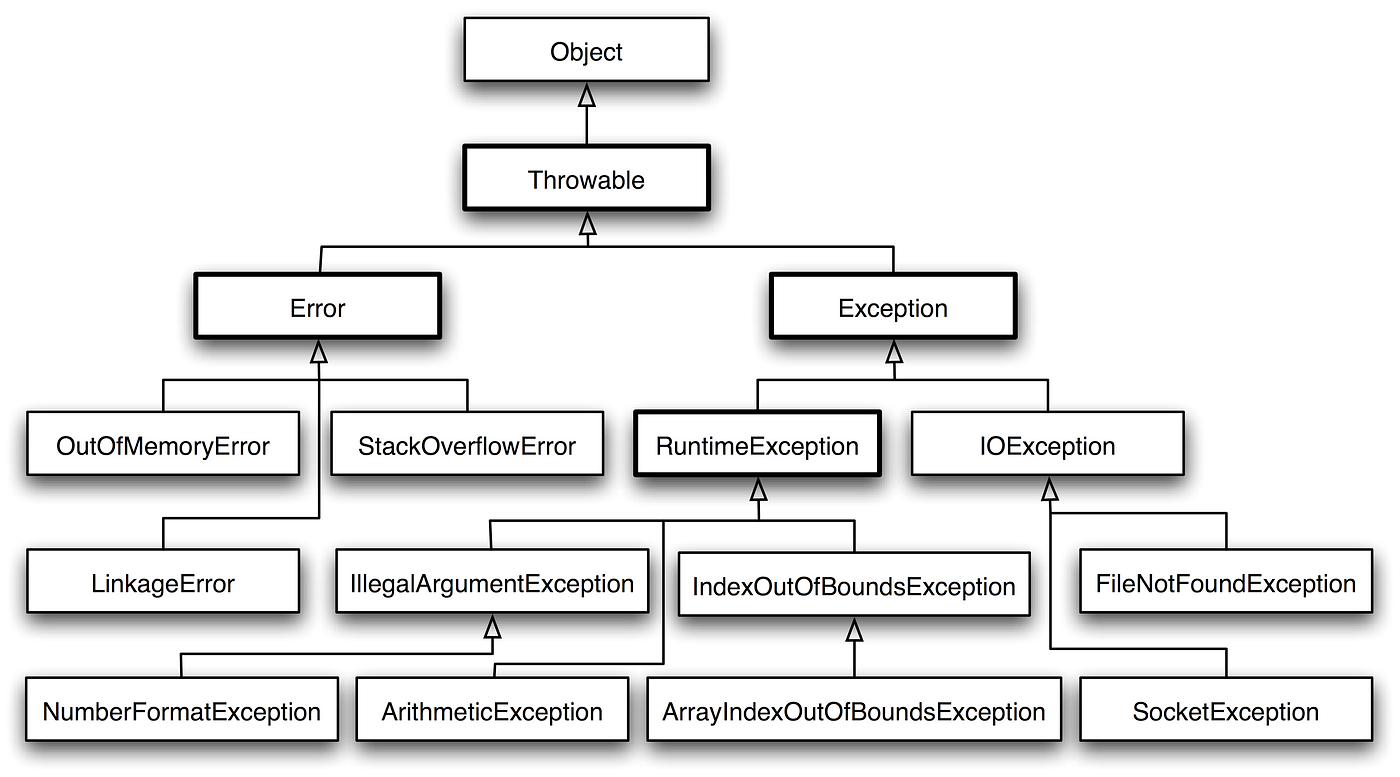
|  |  |  |  |
| --- | --- | --- | --- |
| **Error Type** | **When it Occurs** | **Detected By** | **Example** |
| **Compile-time Error** | During compilation | Java compiler | Syntax error, data type mismatch |
| **Runtime Error** | During program execution | Java Virtual Machine (JVM) | Division by zero, null pointer access |
| **Logical Error** | Program runs but gives wrong output | Developer/Test cases | Incorrect conditions, wrong calculations |

**Exception Handling in Java**

The **Exception Handling in Java** is one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.

**Types of Exceptions:**

* **Checked Exceptions**: These are checked at compile-time. They must be either handled using a try-catch block or declared in the throws clause of the method.
  + Example: IOException, SQLException
* **Unchecked Exceptions**: These are not checked at compile-time, only at runtime. They don't need to be explicitly handled.
  + Example: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException
* **Errors**: Serious issues beyond the control of the program (e.g., OutOfMemoryError). These are usually not handled.



public class Main {

public static void main(String[] args) {

int a = 10;

int b = 0;

try {

// Code that may throw an exception

int result = a / b;

System.out.println("Result: " + result);

} catch (ArithmeticException e) { //Child

// Handling the exception

System.out.println("Cannot divide by zero!");

} catch(Exception e){ //Parent

System.out.println(“Something went wrong”);

}

}

}

**Throw Keyword**

In Java, the **throw** keyword is used to explicitly throw an exception from a method or a block of code.

public class ThrowKeyword {

    public static void main(String[] args) {

        int i = 20;

        int j = 0;

        try {

            j = 18 / i;

            if (j == 0)

                throw new ArithmeticException();

            System.out.println(j);

        } catch (ArithmeticException e) {

            System.out.println("Exception");

        }

        System.out.println("Program executed");

    }

}

**Ducking the exception using Throws Keyword**

**What does "ducking the exception" mean?**

When you use the throws keyword, you're essentially saying, "I'm not going to handle this exception in this method; I'll pass it on to the caller of this method." This is known as **ducking** the exception because you're "ducking out" of the responsibility of handling the exception yourself.

**How does it work?**

When you declare a method using throws, it means that if an exception occurs inside that method, the method won't handle it. Instead, the method that called it will have to handle it (using a try-catch block or propagate it further).

Example :-

Imagine you are a delivery manager for a parcel service. Your job is to hand over the parcels to the customers. However, if a customer is not available, instead of handling the problem yourself (waiting around or deciding what to do), you simply notify your boss that the customer wasn't available. Now, **your boss has to handle the issue** (maybe by deciding to retry delivery later or sending the parcel back to the warehouse).

// Exception class representing the "Customer Not Available" situation

class CustomerNotAvailableException extends Exception {

public CustomerNotAvailableException(String message) {

super(message);

}

}

// Class representing the Delivery Manager

class DeliveryManager {

// Method that tries to deliver a parcel and ducks the exception to the boss

public void deliverParcel(String customerName) throws CustomerNotAvailableException {

if (customerName == null || customerName.isEmpty()) {

// If customer is not available, throw a custom exception

throw new CustomerNotAvailableException("Customer is not available for delivery.");

} else {

System.out.println("Parcel successfully delivered to " + customerName);

}

}

}

// Main class where the boss will handle the delivery

public class DeliveryService {

public static void main(String[] args) {

DeliveryManager manager = new DeliveryManager();

// First attempt: The customer is not available

try {

manager.deliverParcel(""); // Empty string means customer not available

} catch (CustomerNotAvailableException e) {

// Boss handles the issue (maybe retry or send back to the warehouse)

System.out.println("Boss: " + e.getMessage() + " We'll retry delivery later.");

}

// Second attempt: The customer is available

try {

manager.deliverParcel("John Doe"); // Successful delivery

} catch (CustomerNotAvailableException e) {

System.out.println("Boss: " + e.getMessage() + " We'll retry delivery later.");

}

}

}

User Input Using BufferReader and Scanner

**6. System.out.println("Hello World");:**

* **System:** System is a built-in class in Java that provides access to system-level resources, like the console.
* **out:** out is a object of PrintStream inside the System class. Tha
* **println:** This is a method in the **PrintStream** class, which out is an instance of. println prints the text passed to it and then moves the cursor to the next line (hence "print line").
* **"Hello World":** This is a string (a sequence of characters) that will be printed to the console.
* **In this code:** System.out.println("Hello World"); prints the text "Hello World" to the console.

**Try-with-resources Feature in Java**

The **try-with-resources** statement in Java is used to manage resources like file handles, database connections, etc., that need to be closed after their operations. It simplifies resource management by automatically closing resources when they are no longer needed, making the code cleaner and less error-prone.

import java.io.*\**;

public class SimpleTryWithResources {

    public static void main(String[] args) {

*// Using try-with-resources to manage the file resource*

        BufferedReader br = null;

        try {

            InputStreamReader in = new InputStreamReader(System.in);

            br = new BufferedReader(in);

*// Reading the first line of the file*

            String line = br.readLine();

*// Printing the line to the console*

            System.out.println(line);

        } catch (IOException e) {

*// Handling any IO exceptions that may occur*

            System.out.println("Error reading the file: " + e.getMessage());

        } finally {

            br.close();

        }

    }

}

Multithreading in Java

Multithreading is a feature in Java that allows concurrent execution of multiple threads. Each thread represents an independent path of execution. The main goal of multithreading is to make optimal use of the CPU by allowing multiple tasks to be executed simultaneously.

class A extends Thread {

    public void run() {

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

            System.out.println("Hi");

        }

    }

}

class B extends Thread {

    public void run() {

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

            System.out.println("Hello");

        }

    }

}

public class Treading {

    public static void main(String[] args) {

        A obj1 = new A();

        B obj2 = new B();

        obj1.start();

        obj2.start();

        ;

    }

}

**1. Why is the run() method used?**

* The run() method contains the code that defines the task a thread will execute when it is started.
* When you create a thread by extending the Thread class, the thread starts executing whatever is written inside the run() method.

**2. Why is the start() method used?**

* The start() method is used to actually start a new thread.
* When you call start(), it internally calls the run() method in a new thread, allowing the code in run() to be executed concurrently with the main program.

**Thread Priority and Sleep()**

**1. Thread Priority**

* **Thread Priority** determines the relative importance of a thread in terms of CPU time allocation. Threads with higher priority are more likely to get CPU time compared to lower-priority threads. However, thread priority doesn't guarantee the execution order.

**Key Points:**

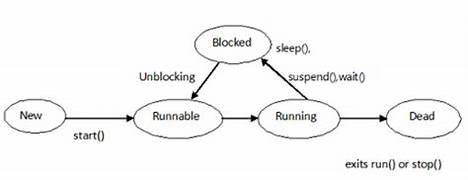
* Every thread in Java has a priority, which is an integer ranging from **MIN\_PRIORITY** (1) to **MAX\_PRIORITY** (10). The default priority is **NORM\_PRIORITY** (5).
* Java thread priorities are just hints to the thread scheduler, which determines when to switch between threads.
* Thread scheduling varies based on the operating system and JVM implementation, so thread priority might not always behave the same on all systems.

**Thread States**

A thread is a program in execution created to perform a specific task. Life cycle of a Java thread starts with its birth and ends on its death.

The start() method of the Thread class is used to initiate the execution of a thread and it goes into runnable state and the sleep() and wait() methods of the Thread class sends the thread into non runnable state.

After non runnable state, thread again comes into runnable state and starts its execution. The run() method of thread is very much important. After executing the run() method, the lifecycle of thread is completed.



**Comparator and Comparable**

**1. Comparable Interface (Used when objects compare themselves)**

Think of **Comparable** as a way to tell Java how an object should compare itself to another object. If you are a **person** and want to say "I am taller than this person," you are **comparable**.

* **When to use?** When the class itself knows how it should be sorted.
* **Example:** Suppose you have a Student class, and you want to sort students by their marks. You can make the Student class implement Comparable and define how one student compares to another student (e.g., by marks).

class Student implements Comparable<Student> {

int marks;

public int compareTo(Student other) {

// Compare students by marks

return this.marks - other.marks; // If positive, 'this' is greater

}

}

* Now, when you sort a list of Student objects, they will automatically be sorted by marks.

**2. Comparator Interface (Used when comparing objects externally)**

**Comparator** is like an external tool that compares two objects. It’s like saying, “I’ll give you a measuring stick, and you tell me how two people compare in height.”

* **When to use?** When you want to compare objects in different ways, but the objects themselves don’t know how to compare.
* **Example:** Imagine you have a list of Student objects, but sometimes you want to sort by marks, and other times by name. You can create different Comparator objects to compare the students in different ways.

Comparator<Student> compareByMarks = new Comparator<Student>() {

public int compare(Student s1, Student s2) {

return s1.marks - s2.marks;

}

};

Comparator<Student> compareByName = new Comparator<Student>() {

public int compare(Student s1, Student s2) {

return s1.name.compareTo(s2.name);

}

};

**Quick Summary:**

* **Comparable**: Object knows how to compare itself (use when sorting by a natural or default order).
* **Comparator**: External tool that tells Java how to compare two objects (use when you need different ways to compare).

**Maven**

[**Maven**](https://maven.apache.org/)**is an open-source**[**build automation**](https://www.browserstack.com/guide/build-automation)**and project management tool widely used for Java applications.  As a build automation tool, it automates the source code compilation and dependency management, assembles binary codes into packages, and executes test scripts. Maven translates and packages your source code so that it becomes an executable application.**

*Let’s learn more about what is Maven in Java. Using Maven, you can create Java deliverables like JAR, EAR, and WAR files. The pom.xml file helps you to do these tasks. These XML files contain your Java project’s name, group ID, and other valuable information. Based on this information, Maven creates the JAR (Java archives) files and transfers them to the Maven repository. You can download and use the JAR files for compilation and testing purposes.*

Maven is a powerful build automation and project management tool primarily used in Java projects. It simplifies the build process by managing dependencies, project configurations, and the lifecycle of the project from compilation to packaging and deployment. Below are key concepts that explain Maven's functionality:

**1. Project Object Model (POM)**

At the heart of every Maven project is the **POM file** (pom.xml). This XML file defines the project structure and contains information about the project’s configuration. It includes:

* **Dependencies**: External libraries or frameworks the project needs (e.g., MySQL Connector, JUnit).
* **Plugins**: Tools to handle tasks like compiling code, running tests, packaging, and deploying the project.
* **Build profiles**: Different settings for different environments (e.g., development, production).

Example pom.xml file:

<project xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0

http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>com.example</groupId>

<artifactId>my-project</artifactId>

<version>1.0.0</version>

<packaging>jar</packaging>

<dependencies>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.33</version>

</dependency>

<!-- More dependencies here -->

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>3.8.1</version>

<configuration>

<source>1.8</source>

<target>1.8</target>

</configuration>

</plugin>

</plugins>

</build>

</project>

**2. Dependency Management**

Maven automatically downloads and manages project dependencies from remote repositories like [Maven Central](https://repo.maven.apache.org/maven2). This ensures that you don't need to manually download JAR files. You declare the dependencies in the POM file, and Maven resolves and downloads them.

**How Dependencies Work:**

* Maven uses a **groupId**, **artifactId**, and **version** to uniquely identify a library.
* For example, for MySQL Connector:

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>8.0.33</version>

</dependency>

* Maven automatically downloads this library from the central repository and includes it in your project.

**3. Maven Lifecycle**

Maven uses a **build lifecycle** that consists of a series of phases, which can be executed in sequence. Some of the important phases are:

* **validate**: Validates the project is correct and all necessary information is available.
* **compile**: Compiles the source code of the project.
* **test**: Runs unit tests against the compiled code using a testing framework like JUnit.
* **package**: Bundles the compiled code into a JAR or WAR file.
* **install**: Installs the packaged code in the local Maven repository (usually ~/.m2 directory).
* **deploy**: Copies the final package to a remote repository, ready to be shared with other developers.

**4. Maven Plugins**

Maven uses plugins to perform specific tasks during the build process. Some important plugins are:

* **maven-compiler-plugin**: Used to compile Java source code.
* **maven-surefire-plugin**: Used to run unit tests.
* **maven-jar-plugin**: Packages the project into a JAR file.

Plugins can be customized in the pom.xml to tailor the build process to your project's needs.

**5. Repositories**

Maven uses two types of repositories to store libraries:

* **Local Repository**: Stored on your local machine, typically located at ~/.m2/repository. When you build a project, Maven first checks this location for dependencies.
* **Remote Repository**: If a dependency is not available in the local repository, Maven downloads it from a remote repository, such as **Maven Central** or a company’s internal repository.

**6. Maven Coordinates**

Each Maven project and dependency is uniquely identified by **coordinates** that consist of:

* **groupId**: Typically represents the organization or group that created the project (e.g., org.apache.maven).
* **artifactId**: The name of the project or library (e.g., maven-compiler-plugin).
* **version**: The version of the project (e.g., 3.8.1).

**7. Maven Archetypes**

Maven provides **archetypes**—templates for generating new projects. You can quickly bootstrap a new project using an archetype. For example, to create a new Java project, you can use the Maven command:

bash

Copy code

mvn archetype:generate -DgroupId=com.example -DartifactId=my-app -DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false

This generates a basic Java project structure with a sample pom.xml file.

**8. Maven Goals**

Maven commands are called **goals**, which correspond to specific tasks. For example:

* mvn clean: Deletes the target directory and any compiled files.
* mvn compile: Compiles the project source code.
* mvn package: Packages the compiled code into a JAR or WAR file.
* mvn install: Installs the package into the local repository.

**9. Transitive Dependencies**

Maven automatically resolves **transitive dependencies**, meaning it will also include the dependencies of your dependencies in the build. For instance, if your project requires Library A, and Library A requires Library B, Maven will include both Library A and Library B in your project without any additional configuration.

**JDBC (Java Database Connectivity)**

**JDBC Setup**

**1.Import Packages (java.sql)**

To work with JDBC, we need to import the necessary classes from the java.sql package. These classes include Connection, DriverManager, Statement, and ResultSet.

2. **Load Drivers**

* JDBC drivers are necessary to connect to different databases (MySQL, Oracle, etc.). Depending on the database, the driver class needs to be loaded into memory before you can use JDBC.
* **Modern Approach** (from JDBC 4.0+):
  + The driver is automatically loaded by the DriverManager if it's included in the classpath.
* **Legacy Approach** (before JDBC 4.0):
  + You need to manually load the driver using Class.forName() method.

3. **Register Driver**

* This step is automatically handled in newer JDBC versions when the driver is loaded using the DriverManager. However, in older versions, you had to explicitly register the driver to the DriverManager.
* **Modern Approach**: DriverManager handles registration when Class.forName() is called or if the driver is in the classpath.

4. **Create Connection**

* After loading and registering the driver, you need to create a connection to the database using DriverManager. You typically provide a URL, username, and password for the database.

5. **Create Statement**

* Once the connection is established, you can create a Statement object that will be used to execute SQL queries. There are three types of statements:
  + Statement: For simple SQL queries without parameters.
  + PreparedStatement: For precompiled queries with parameters.
  + CallableStatement: For executing stored procedures.

6. **Execute Statement**

* Using the Statement object, you can execute SQL queries like SELECT, INSERT, UPDATE, DELETE, etc.
* For SELECT queries, you get a ResultSet object to iterate over the results.

7. **Close**

* It is important to close the Connection, Statement, and ResultSet objects to avoid resource leaks.
* Always close these resources in the reverse order of their creation (ResultSet → Statement → Connection). Using try-with-resources is recommended in Java 7+.

Code

// Step 1: Import Packages

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.Statement;

import java.sql.ResultSet;

import java.sql.SQLException;

public class JDBCDemo {

public static void main(String[] args) {

String url = "jdbc:mysql://localhost:3306/myDatabase";

String username = "root";

String password = "password";

try {

// Step 2: Load driver (optional for JDBC 4.0+)

Class.forName("com.mysql.cj.jdbc.Driver");

// Step 3: Create connection

Connection conn = DriverManager.getConnection(url, username, password);

// Step 4: Create statement

Statement stmt = conn.createStatement();

// Step 5: Execute query

String query = "SELECT \* FROM users";

ResultSet rs = stmt.executeQuery(query);

// Step 6: Process the result set

while (rs.next()) {

System.out.println("User: " + rs.getString("username"));

}

// Step 7: Close resources

rs.close();

stmt.close();

conn.close();

} catch (SQLException | ClassNotFoundException e) {

e.printStackTrace();

}

}}