**1. Basic Structure**

Every Java program starts with a class definition and a main method.

java

Copy code

public class MyClass {

public static void main(String[] args) {

// Code execution starts here

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

}

}

* **Class Definition**: public class MyClass defines a class named MyClass.
* **Main Method**: public static void main(String[] args) is the entry point of any Java application.
* **Comments**:
  + // Single-line comment
  + /\* Multi-line comment \*/
  + /\*\* Javadoc comment \*/

**2. Data Types**

Java has two main categories of data types: **Primitive** and **Non-Primitive**.

**Primitive Types**

These are basic types built into the language.

| **Type** | **Size (bits)** | **Description** |
| --- | --- | --- |
| byte | 8 | Integer (-128 to 127) |
| short | 16 | Integer (-32,768 to 32,767) |
| int | 32 | Integer |
| long | 64 | Integer |
| float | 32 | Floating-point number |
| double | 64 | Double-precision number |
| char | 16 | Single Unicode character |
| boolean | - | true or false |

**Non-Primitive Types**

These are objects and can be used to call methods.

* **Strings**: String, which is a sequence of characters.
* **Arrays**: Fixed-size collections of elements.
* **Classes**: User-defined types.
* **Interfaces**: Abstract types used to specify behaviors.

**3. Variable Declaration**

Variables store data values. They must be declared with a data type before use.

java

Copy code

// Declaration and Initialization

int a = 5;

float b = 5.67f; // 'f' denotes float literal

char c = 'A';

boolean isTrue = true;

String str = "Hello, World!";

// Multiple Declarations

int x = 10, y = 20, z = 30;

**Constants**

Use final keyword to declare constants whose values cannot be changed.

java

Copy code

final double PI = 3.14159;

**4. Control Structures**

Control the flow of the program using conditional statements and loops.

**4.1. If-Else Statements**

java

Copy code

if (condition) {

// Executes if condition is true

} else if (anotherCondition) {

// Executes if anotherCondition is true

} else {

// Executes if none of the above conditions are true

}

**4.2. Switch Statement**

Efficiently handle multiple conditions based on a single variable.

java

Copy code

int day = 2;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

// Additional cases

default:

System.out.println("Invalid day");

}

**4.3. Loops**

**For Loop**

Used when the number of iterations is known.

java

Copy code

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

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

}

**Enhanced For Loop (For-Each)**

Ideal for iterating over arrays and collections.

java

Copy code

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

for (int num : numbers) {

System.out.println(num);

}

**While Loop**

Used when the number of iterations is not known beforehand.

java

Copy code

int i = 0;

while (i < 5) {

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

i++;

}

**Do-While Loop**

Executes the block at least once before checking the condition.

java

Copy code

int i = 0;

do {

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

i++;

} while (i < 5);

**5. Arrays**

Arrays store multiple values of the same type.

**Declaration and Instantiation**

java

Copy code

int[] arr = new int[5]; // Creates an array of 5 integers

**Initialization**

java

Copy code

int[] arr = {1, 2, 3, 4, 5}; // Initializes the array with values

**Accessing Elements**

java

Copy code

System.out.println(arr[0]); // Outputs: 1

arr[2] = 10; // Updates the third element to 10

**Multidimensional Arrays**

java

Copy code

int[][] matrix = {

{1, 2, 3},

{4, 5, 6},

{7, 8, 9}

};

System.out.println(matrix[1][2]); // Outputs: 6

**6. Methods (Functions)**

Methods encapsulate reusable code blocks.

**Method Declaration**

java

Copy code

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

return a + b;

}

**Method Invocation**

java

Copy code

public static void main(String[] args) {

int result = add(5, 3);

System.out.println("Sum: " + result); // Output: Sum: 8

}

**Method Overloading**

Multiple methods with the same name but different parameters.

java

Copy code

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

return a + b;

}

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

return a + b;

}

**7. Object-Oriented Programming (OOP)**

Java is an object-oriented language that uses classes and objects to model real-world entities.

**7.1. Class and Object**

**Class Definition**

java

Copy code

class Animal {

String name;

int age;

void speak() {

System.out.println(name + " makes a sound.");

}

}

**Creating Objects**

java

Copy code

public class Main {

public static void main(String[] args) {

Animal dog = new Animal();

dog.name = "Buddy";

dog.age = 5;

dog.speak(); // Output: Buddy makes a sound.

}

}

**7.2. Constructors**

Special methods to initialize new objects.

java

Copy code

class Car {

String model;

int year;

// Constructor

Car(String model, int year) {

this.model = model;

this.year = year;

}

void displayInfo() {

System.out.println(model + " - " + year);

}

}

public class Main {

public static void main(String[] args) {

Car myCar = new Car("Toyota", 2020);

myCar.displayInfo(); // Output: Toyota - 2020

}

}

**7.3. Inheritance**

Allows one class to inherit properties and methods from another.

java

Copy code

class Animal {

void eat() {

System.out.println("This animal eats food.");

}

}

class Dog extends Animal {

void bark() {

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

}

}

public class Main {

public static void main(String[] args) {

Dog dog = new Dog();

dog.eat(); // Inherited method

dog.bark(); // Own method

}

}

**7.4. Polymorphism**

Ability of objects to take on many forms.

**Compile-Time Polymorphism (Method Overloading)**

java

Copy code

class MathUtils {

int multiply(int a, int b) {

return a \* b;

}

double multiply(double a, double b) {

return a \* b;

}

}

**Run-Time Polymorphism (Method Overriding)**

java

Copy code

class Animal {

void sound() {

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

}

}

class Cat extends Animal {

@Override

void sound() {

System.out.println("Meow");

}

}

public class Main {

public static void main(String[] args) {

Animal myCat = new Cat();

myCat.sound(); // Output: Meow

}

}

**7.5. Encapsulation**

Encapsulating data by using private variables and public getter/setter methods.

java

Copy code

class Person {

private String name;

private int age;

// Getter

public String getName() {

return name;

}

// Setter

public void setName(String name) {

this.name = name;

}

// Getter

public int getAge() {

return age;

}

// Setter

public void setAge(int age) {

if(age > 0) {

this.age = age;

}

}

}

**7.6. Abstraction**

Hiding complex implementation details and showing only the necessary features.

**Abstract Class**

java

Copy code

abstract class Shape {

abstract double area();

void display() {

System.out.println("Displaying shape.");

}

}

class Circle extends Shape {

double radius;

Circle(double radius) {

this.radius = radius;

}

@Override

double area() {

return Math.PI \* radius \* radius;

}

}

**Interface**

java

Copy code

interface Drawable {

void draw();

}

class Rectangle implements Drawable {

@Override

public void draw() {

System.out.println("Drawing a rectangle.");

}

}

**8. Exception Handling**

Manage errors gracefully using try-catch blocks.

java

Copy code

try {

int result = 10 / 0; // This will throw ArithmeticException

} catch (ArithmeticException e) {

System.out.println("Arithmetic error: " + e.getMessage());

} catch (Exception e) {

System.out.println("An error occurred: " + e.getMessage());

} finally {

System.out.println("This block always executes.");

}

**Custom Exceptions**

java

Copy code

class InvalidAgeException extends Exception {

public InvalidAgeException(String message) {

super(message);

}

}

public class Main {

static void validateAge(int age) throws InvalidAgeException {

if(age < 18) {

throw new InvalidAgeException("Age must be at least 18.");

}

}

public static void main(String[] args) {

try {

validateAge(16);

} catch (InvalidAgeException e) {

System.out.println(e.getMessage()); // Output: Age must be at least 18.

}

}

}

**9. File Handling**

Perform read and write operations on files.

java

Copy code

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Scanner;

public class FileHandlingExample {

public static void main(String[] args) {

try {

// Create a file

File myFile = new File("example.txt");

if (myFile.createNewFile()) {

System.out.println("File created: " + myFile.getName());

}

// Write to file

FileWriter writer = new FileWriter("example.txt");

writer.write("Hello, Java File Handling!");

writer.close();

System.out.println("Successfully wrote to the file.");

// Read from file

Scanner reader = new Scanner(myFile);

while (reader.hasNextLine()) {

String data = reader.nextLine();

System.out.println("File Content: " + data);

}

reader.close();

} catch (IOException e) {

System.out.println("An error occurred.");

e.printStackTrace();

}

}

}

**BufferedReader and BufferedWriter**

More efficient for reading and writing large files.

java

Copy code

import java.io.\*;

public class BufferedExample {

public static void main(String[] args) {

// Writing with BufferedWriter

try (BufferedWriter bw = new BufferedWriter(new FileWriter("buffered.txt"))) {

bw.write("Buffered Writer Example");

} catch (IOException e) {

e.printStackTrace();

}

// Reading with BufferedReader

try (BufferedReader br = new BufferedReader(new FileReader("buffered.txt"))) {

String line;

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

System.out.println("BufferedReader: " + line);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

**10. Commonly Used Classes**

**10.1. String Methods**

Strings are immutable objects in Java with various useful methods.

java

Copy code

String str = "Hello, World!";

System.out.println(str.length()); // Outputs: 13

System.out.println(str.toUpperCase()); // Outputs: HELLO, WORLD!

System.out.println(str.toLowerCase()); // Outputs: hello, world!

System.out.println(str.contains("Hello")); // Outputs: true

System.out.println(str.substring(7)); // Outputs: World!

System.out.println(str.replace("World", "Java")); // Outputs: Hello, Java!

System.out.println(str.charAt(1)); // Outputs: e

**10.2. StringBuilder and StringBuffer**

Mutable sequences of characters, useful for efficient string manipulation.

java

Copy code

// StringBuilder Example

StringBuilder sb = new StringBuilder("Hello");

sb.append(", Java!");

System.out.println(sb.toString()); // Outputs: Hello, Java!

// StringBuffer Example (thread-safe)

StringBuffer sbf = new StringBuffer("Hello");

sbf.append(", Buffer!");

System.out.println(sbf.toString()); // Outputs: Hello, Buffer!

**10.3. ArrayList**

Resizable array implementation from the Collections Framework.

java

Copy code

import java.util.ArrayList;

public class ArrayListExample {

public static void main(String[] args) {

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

// Adding elements

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Cherry");

// Accessing elements

System.out.println(fruits.get(1)); // Output: Banana

// Removing elements

fruits.remove("Banana");

// Iterating through ArrayList

for(String fruit : fruits) {

System.out.println(fruit);

}

// Size of ArrayList

System.out.println("Total Fruits: " + fruits.size()); // Output: Total Fruits: 2

}

}

**10.4. HashMap**

Stores key-value pairs for efficient data retrieval.

java

Copy code

import java.util.HashMap;

public class HashMapExample {

public static void main(String[] args) {

HashMap<Integer, String> map = new HashMap<>();

// Adding key-value pairs

map.put(1, "Java");

map.put(2, "Python");

map.put(3, "C++");

// Accessing value by key

System.out.println(map.get(2)); // Output: Python

// Removing a key-value pair

map.remove(3);

// Iterating through HashMap

for(Integer key : map.keySet()) {

System.out.println("Key: " + key + ", Value: " + map.get(key));

}

// Size of HashMap

System.out.println("Total Entries: " + map.size()); // Output: Total Entries: 2

}

}

**10.5. LinkedList**

A doubly-linked list implementation from the Collections Framework.

java

Copy code

import java.util.LinkedList;

public class LinkedListExample {

public static void main(String[] args) {

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

// Adding elements

list.add("First");

list.add("Second");

list.addFirst("Zero");

list.addLast("Third");

// Accessing elements

System.out.println(list.get(2)); // Output: Second

// Removing elements

list.remove("Second");

// Iterating through LinkedList

for(String item : list) {

System.out.println(item);

}

// Size of LinkedList

System.out.println("Total Items: " + list.size()); // Output: Total Items: 3

}

}

**10.6. Stack and Queue**

Data structures for LIFO and FIFO operations respectively.

java

Copy code

import java.util.Stack;

import java.util.LinkedList;

import java.util.Queue;

public class StackQueueExample {

public static void main(String[] args) {

// Stack Example

Stack<String> stack = new Stack<>();

stack.push("First");

stack.push("Second");

System.out.println(stack.pop()); // Output: Second

// Queue Example

Queue<String> queue = new LinkedList<>();

queue.add("First");

queue.add("Second");

System.out.println(queue.remove()); // Output: First

}

}

**11. Multithreading**

Java supports concurrent execution using threads.

**11.1. Extending Thread Class**

java

Copy code

class MyThread extends Thread {

public void run() {

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

System.out.println("Thread running: " + i);

}

}

}

public class ThreadExample {

public static void main(String[] args) {

MyThread t1 = new MyThread();

t1.start(); // Starts the new thread

}

}

**11.2. Implementing Runnable Interface**

java

Copy code

class MyRunnable implements Runnable {

public void run() {

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

System.out.println("Runnable running: " + i);

}

}

}

public class RunnableExample {

public static void main(String[] args) {

Thread t1 = new Thread(new MyRunnable());

t1.start();

}

}

**11.3. Synchronization**

Ensures that multiple threads do not interfere with each other.

java

Copy code

class Counter {

private int count = 0;

// Synchronized method

public synchronized void increment() {

count++;

}

public int getCount() {

return count;

}

}

public class SyncExample {

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

Counter counter = new Counter();

Runnable r = () -> {

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

counter.increment();

}

};

Thread t1 = new Thread(r);

Thread t2 = new Thread(r);

t1.start();

t2.start();

t1.join();

t2.join();

System.out.println("Final Count: " + counter.getCount()); // Output: Final Count: 2000

}

}

**11.4. Thread Pools (ExecutorService)**

Manages a pool of threads for executing tasks.

java

Copy code

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

public class ThreadPoolExample {

public static void main(String[] args) {

ExecutorService executor = Executors.newFixedThreadPool(2); // Pool of 2 threads

Runnable task1 = () -> {

System.out.println("Task1 is running");

};

Runnable task2 = () -> {

System.out.println("Task2 is running");

};

executor.execute(task1);

executor.execute(task2);

executor.shutdown(); // Initiates an orderly shutdown

}

}

**12. Lambda Expressions (Java 8)**

Provides a clear and concise way to represent a single method interface using an expression.

java

Copy code

interface Drawable {

void draw();

}

public class LambdaExample {

public static void main(String[] args) {

// Using Lambda Expression

Drawable d = () -> System.out.println("Drawing with Lambda");

d.draw(); // Output: Drawing with Lambda

}

}

**Lambda with Parameters**

java

Copy code

interface Adder {

int add(int a, int b);

}

public class LambdaWithParams {

public static void main(String[] args) {

Adder add = (a, b) -> a + b;

System.out.println("Sum: " + add.add(5, 3)); // Output: Sum: 8

}

}

**Lambda with Multiple Statements**

java

Copy code

interface Calculator {

int calculate(int a, int b);

}

public class MultiStatementLambda {

public static void main(String[] args) {

Calculator calc = (a, b) -> {

int sum = a + b;

int product = a \* b;

return sum + product;

};

System.out.println("Result: " + calc.calculate(2, 3)); // Output: Result: 11

}

}

**13. Streams (Java 8)**

Provides a functional approach to processing sequences of elements.

java

Copy code

import java.util.Arrays;

import java.util.List;

public class StreamExample {

public static void main(String[] args) {

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

// Filter even numbers and print them

numbers.stream()

.filter(n -> n % 2 == 0)

.forEach(System.out::println); // Output: 2 4

// Map: Square each number and collect to a list

List<Integer> squares = numbers.stream()

.map(n -> n \* n)

.toList();

System.out.println(squares); // Output: [1, 4, 9, 16, 25]

// Reduce: Sum of all numbers

int sum = numbers.stream()

.reduce(0, Integer::sum);

System.out.println("Sum: " + sum); // Output: Sum: 15

}

}

**Parallel Streams**

Utilize multiple threads for faster processing.

java

Copy code

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

numbers.parallelStream()

.filter(n -> n % 2 == 0)

.forEach(System.out::println); // May execute in parallel

**14. Interfaces and Abstract Classes**

**14.1. Interfaces**

Define a contract that implementing classes must follow.

java

Copy code

interface Vehicle {

void start();

void stop();

}

class Car implements Vehicle {

@Override

public void start() {

System.out.println("Car started.");

}

@Override

public void stop() {

System.out.println("Car stopped.");

}

}

public class InterfaceExample {

public static void main(String[] args) {

Vehicle car = new Car();

car.start(); // Output: Car started.

car.stop(); // Output: Car stopped.

}

}

**14.2. Abstract Classes**

Can have both abstract and concrete methods.

java

Copy code

abstract class Appliance {

abstract void turnOn();

void plugIn() {

System.out.println("Appliance plugged in.");

}

}

class WashingMachine extends Appliance {

@Override

void turnOn() {

System.out.println("Washing Machine is now ON.");

}

}

public class AbstractClassExample {

public static void main(String[] args) {

WashingMachine wm = new WashingMachine();

wm.plugIn(); // Output: Appliance plugged in.

wm.turnOn(); // Output: Washing Machine is now ON.

}

}

**15. Packages**

Organize classes into namespaces to avoid naming conflicts and to control access.

**15.1. Creating a Package**

java

Copy code

// File: com/example/utils/MathUtils.java

package com.example.utils;

public class MathUtils {

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

return a + b;

}

}

**15.2. Using a Package**

java

Copy code

// File: com/example/app/MainApp.java

package com.example.app;

import com.example.utils.MathUtils;

public class MainApp {

public static void main(String[] args) {

int sum = MathUtils.add(10, 20);

System.out.println("Sum: " + sum); // Output: Sum: 30

}

}

**15.3. Access Modifiers and Packages**

* **public**: Accessible from any other class.
* **protected**: Accessible within the same package and subclasses.
* **default (no modifier)**: Accessible only within the same package.
* **private**: Accessible only within the declared class.

**16. Access Modifiers**

Control the visibility of classes, methods, and variables.

| **Modifier** | **Class** | **Package** | **Subclass** | **World** |
| --- | --- | --- | --- | --- |
| public | Yes | Yes | Yes | Yes |
| protected | No | Yes | Yes | No |
| default | No | Yes | No | No |
| private | No | No | No | No |

**Example**

java

Copy code

public class AccessModifiersExample {

public int publicVar = 1;

protected int protectedVar = 2;

int defaultVar = 3; // default

private int privateVar = 4;

public void publicMethod() {}

protected void protectedMethod() {}

void defaultMethod() {}

private void privateMethod() {}

}

**17. Generics**

Enable classes, interfaces, and methods to operate on types specified by the programmer.

**Generic Class**

java

Copy code

class Box<T> {

private T item;

public void set(T item) { this.item = item; }

public T get() { return item; }

}

public class GenericsExample {

public static void main(String[] args) {

Box<String> stringBox = new Box<>();

stringBox.set("Hello Generics");

System.out.println(stringBox.get()); // Output: Hello Generics

Box<Integer> intBox = new Box<>();

intBox.set(100);

System.out.println(intBox.get()); // Output: 100

}

}

**Generic Methods**

java

Copy code

public class GenericMethods {

public static <T> void printArray(T[] array) {

for(T element : array) {

System.out.println(element);

}

}

public static void main(String[] args) {

Integer[] nums = {1, 2, 3};

String[] words = {"Java", "Generics"};

printArray(nums);

printArray(words);

}

}

**18. Enums**

Define a fixed set of constants.

java

Copy code

enum Day {

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

}

public class EnumExample {

public static void main(String[] args) {

Day today = Day.WEDNESDAY;

switch(today) {

case MONDAY:

System.out.println("Start of the week.");

break;

case FRIDAY:

System.out.println("End of the work week.");

break;

default:

System.out.println("Midweek day.");

break;

}

}

}

**Enum with Methods**

java

Copy code

enum Operation {

ADD, SUBTRACT, MULTIPLY, DIVIDE;

public double apply(double a, double b) {

switch(this) {

case ADD: return a + b;

case SUBTRACT: return a - b;

case MULTIPLY: return a \* b;

case DIVIDE: return a / b;

default: throw new AssertionError("Unknown operation: " + this);

}

}

}

public class EnumWithMethods {

public static void main(String[] args) {

double result = Operation.ADD.apply(5, 3);

System.out.println("Result: " + result); // Output: Result: 8.0

}

}

**19. Static and Final Keywords**

**19.1. Static Keyword**

* **Static Variables**: Shared among all instances of a class.
* **Static Methods**: Belong to the class, not instances.
* **Static Blocks**: Execute when the class is loaded.

java

Copy code

class Counter {

static int count = 0;

Counter() {

count++;

}

static void displayCount() {

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

}

}

public class StaticExample {

public static void main(String[] args) {

Counter c1 = new Counter();

Counter c2 = new Counter();

Counter.displayCount(); // Output: Count: 2

}

}

**19.2. Final Keyword**

* **Final Variables**: Constants that cannot be modified.
* **Final Methods**: Cannot be overridden by subclasses.
* **Final Classes**: Cannot be subclassed.

java

Copy code

final class Constants {

public static final double PI = 3.14159;

}

// Attempting to extend a final class will result in a compile-time error

// class ExtendedConstants extends Constants {} // Error

class Base {

public final void show() {

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

}

}

class Derived extends Base {

// Cannot override final method

// public void show() { System.out.println("Override."); } // Error

}

**20. Inner Classes**

Classes defined within another class.

**20.1. Member Inner Class**

java

Copy code

class OuterClass {

private String message = "Hello from Outer Class";

class InnerClass {

void display() {

System.out.println(message);

}

}

}

public class InnerClassExample {

public static void main(String[] args) {

OuterClass outer = new OuterClass();

OuterClass.InnerClass inner = outer.new InnerClass();

inner.display(); // Output: Hello from Outer Class

}

}

**20.2. Static Nested Class**

java

Copy code

class OuterClass {

static class StaticNestedClass {

void display() {

System.out.println("Inside Static Nested Class");

}

}

}

public class StaticNestedExample {

public static void main(String[] args) {

OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();

nested.display(); // Output: Inside Static Nested Class

}

}

**20.3. Anonymous Inner Class**

Used for implementing interfaces or extending classes on the fly.

java

Copy code

interface Greeting {

void greet();

}

public class AnonymousInnerClassExample {

public static void main(String[] args) {

Greeting greeting = new Greeting() {

@Override

public void greet() {

System.out.println("Hello from Anonymous Inner Class");

}

};

greeting.greet(); // Output: Hello from Anonymous Inner Class

}

}

**21. Recursion**

A method calling itself to solve a problem.

java

Copy code

public class RecursionExample {

// Factorial using recursion

static int factorial(int n) {

if(n == 0) return 1;

return n \* factorial(n - 1);

}

public static void main(String[] args) {

int result = factorial(5);

System.out.println("Factorial: " + result); // Output: Factorial: 120

}

}

**22. Regular Expressions (Regex)**

Patterns used to match character combinations in strings.

java

Copy code

import java.util.regex.\*;

public class RegexExample {

public static void main(String[] args) {

String text = "Java Regex Example 123";

// Pattern to find digits

Pattern pattern = Pattern.compile("\\d+");

Matcher matcher = pattern.matcher(text);

while(matcher.find()) {

System.out.println("Found: " + matcher.group()); // Output: Found: 123

}

// Validate email

String email = "example@test.com";

String emailRegex = "^[A-Za-z0-9+\_.-]+@(.+)$";

boolean isValid = email.matches(emailRegex);

System.out.println("Is valid email: " + isValid); // Output: Is valid email: true

}

}

**23. Serialization**

Converting an object into a byte stream for storage or transmission.

**23.1. Serializable Interface**

java

Copy code

import java.io.\*;

class Person implements Serializable {

private static final long serialVersionUID = 1L;

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

}

public class SerializationExample {

public static void main(String[] args) {

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

// Serialize the object

try (FileOutputStream fileOut = new FileOutputStream("person.ser");

ObjectOutputStream out = new ObjectOutputStream(fileOut)) {

out.writeObject(person);

System.out.println("Serialized data is saved in person.ser");

} catch (IOException i) {

i.printStackTrace();

}

// Deserialize the object

person = null;

try (FileInputStream fileIn = new FileInputStream("person.ser");

ObjectInputStream in = new ObjectInputStream(fileIn)) {

person = (Person) in.readObject();

System.out.println("Deserialized Person:");

System.out.println("Name: " + person.name); // Output: Name: Alice

System.out.println("Age: " + person.age); // Output: Age: 30

} catch (IOException | ClassNotFoundException i) {

i.printStackTrace();

}

}

}

**23.2. transient Keyword**

Fields marked as transient are not serialized.

java

Copy code

class User implements Serializable {

String username;

transient String password; // Will not be serialized

User(String username, String password) {

this.username = username;

this.password = password;

}

}

**24. JDBC (Java Database Connectivity)**

Connect and execute queries with databases.

**24.1. Basic JDBC Steps**

1. **Load JDBC Driver**
2. **Establish Connection**
3. **Create Statement**
4. **Execute Query**
5. **Process Results**
6. **Close Connection**

**24.2. Example with MySQL**

java

Copy code

import java.sql.\*;

public class JDBCDemo {

public static void main(String[] args) {

String jdbcURL = "jdbc:mysql://localhost:3306/mydatabase";

String username = "root";

String password = "password";

try {

// 1. Load JDBC Driver (optional for newer versions)

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

// 2. Establish Connection

Connection connection = DriverManager.getConnection(jdbcURL, username, password);

System.out.println("Connected to the database.");

// 3. Create Statement

Statement statement = connection.createStatement();

// 4. Execute Query

String sql = "SELECT id, name FROM users";

ResultSet resultSet = statement.executeQuery(sql);

// 5. Process Results

while(resultSet.next()) {

int id = resultSet.getInt("id");

String name = resultSet.getString("name");

System.out.println("ID: " + id + ", Name: " + name);

}

// 6. Close Connection

connection.close();

} catch (SQLException | ClassNotFoundException e) {

e.printStackTrace();

}

}

}

**24.3. Using PreparedStatement**

Prevents SQL injection and improves performance for repeated queries.

java

Copy code

String insertSQL = "INSERT INTO users (name, email) VALUES (?, ?)";

PreparedStatement pstmt = connection.prepareStatement(insertSQL);

pstmt.setString(1, "John Doe");

pstmt.setString(2, "john@example.com");

int rowsInserted = pstmt.executeUpdate();

System.out.println("Rows Inserted: " + rowsInserted);

**25. Design Patterns**

Common solutions to recurring design problems.

**25.1. Singleton Pattern**

Ensures a class has only one instance and provides a global point of access.

java

Copy code

public class Singleton {

private static Singleton instance;

// Private constructor to prevent instantiation

private Singleton() {}

// Public method to provide access to the instance

public static Singleton getInstance() {

if(instance == null) {

instance = new Singleton();

}

return instance;

}

public void showMessage() {

System.out.println("Hello from Singleton!");

}

}

public class SingletonDemo {

public static void main(String[] args) {

Singleton single = Singleton.getInstance();

single.showMessage(); // Output: Hello from Singleton!

}

}

**25.2. Factory Pattern**

Creates objects without specifying the exact class of the object.

java

Copy code

// Product Interface

interface Shape {

void draw();

}

// Concrete Products

class Circle implements Shape {

public void draw() {

System.out.println("Drawing Circle.");

}

}

class Square implements Shape {

public void draw() {

System.out.println("Drawing Square.");

}

}

// Factory Class

class ShapeFactory {

public Shape getShape(String shapeType) {

if(shapeType == null) return null;

if(shapeType.equalsIgnoreCase("CIRCLE")) return new Circle();

if(shapeType.equalsIgnoreCase("SQUARE")) return new Square();

return null;

}

}

// Client

public class FactoryPatternDemo {

public static void main(String[] args) {

ShapeFactory factory = new ShapeFactory();

Shape shape1 = factory.getShape("CIRCLE");

shape1.draw(); // Output: Drawing Circle.

Shape shape2 = factory.getShape("SQUARE");

shape2.draw(); // Output: Drawing Square.

}

}

**25.3. Observer Pattern**

Defines a one-to-many dependency so that when one object changes state, all its dependents are notified.

java

Copy code

import java.util.ArrayList;

import java.util.List;

// Observer Interface

interface Observer {

void update(String message);

}

// Subject Class

class Subject {

private List<Observer> observers = new ArrayList<>();

void addObserver(Observer o) {

observers.add(o);

}

void removeObserver(Observer o) {

observers.remove(o);

}

void notifyObservers(String message) {

for(Observer o : observers) {

o.update(message);

}

}

}

// Concrete Observer

class ConcreteObserver implements Observer {

private String name;

ConcreteObserver(String name) {

this.name = name;

}

public void update(String message) {

System.out.println(name + " received: " + message);

}

}

// Demo

public class ObserverPatternDemo {

public static void main(String[] args) {

Subject subject = new Subject();

Observer obs1 = new ConcreteObserver("Observer1");

Observer obs2 = new ConcreteObserver("Observer2");

subject.addObserver(obs1);

subject.addObserver(obs2);

subject.notifyObservers("Hello Observers!");

// Output:

// Observer1 received: Hello Observers!

// Observer2 received: Hello Observers!

}

}

**26. Annotations**

Provide metadata about the program and can be used by the compiler or at runtime.

**26.1. Built-in Annotations**

* @Override: Indicates that a method overrides a method in a superclass.
* @Deprecated: Marks a method as deprecated.
* @SuppressWarnings: Suppresses compiler warnings.

java

Copy code

class Parent {

void display() {

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

}

}

class Child extends Parent {

@Override

void display() { // Correct usage

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

}

@Deprecated

void oldMethod() {

// Deprecated method

}

}

**26.2. Custom Annotations**

java

Copy code

import java.lang.annotation.\*;

@Retention(RetentionPolicy.RUNTIME)

@Target(ElementType.METHOD)

@interface CustomAnnotation {

String value();

}

class TestClass {

@CustomAnnotation("Testing")

public void testMethod() {

System.out.println("Method with Custom Annotation");

}

}

public class CustomAnnotationExample {

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

TestClass obj = new TestClass();

obj.testMethod();

// Accessing annotation

Method method = TestClass.class.getMethod("testMethod");

if(method.isAnnotationPresent(CustomAnnotation.class)) {

CustomAnnotation annotation = method.getAnnotation(CustomAnnotation.class);

System.out.println("Annotation value: " + annotation.value()); // Output: Annotation value: Testing

}

}

}

**27. Java APIs**

Java provides a vast set of APIs for various functionalities. Here are some commonly used ones:

**27.1. java.lang**

* Fundamental classes like String, Math, System, Object, etc.

**27.2. java.util**

* Collections Framework (ArrayList, HashMap, etc.)
* Date and Time (Date, Calendar, GregorianCalendar)
* Utility classes (Scanner, Random, etc.)

**27.3. java.io**

* Input and Output through streams (File, FileReader, FileWriter, etc.)

**27.4. java.nio**

* Non-blocking I/O operations.

**27.5. java.net**

* Networking capabilities (Socket, ServerSocket, etc.)

**27.6. java.sql**

* Database connectivity and operations.

**27.7. java.time (Java 8+)**

* Modern Date and Time API (LocalDate, LocalTime, LocalDateTime, etc.)

java

Copy code

import java.time.LocalDate;

import java.time.format.DateTimeFormatter;

public class DateTimeExample {

public static void main(String[] args) {

LocalDate today = LocalDate.now();

System.out.println("Today's Date: " + today); // Output: Today's Date: 2024-04-27

// Formatting date

DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd-MM-yyyy");

String formattedDate = today.format(formatter);

System.out.println("Formatted Date: " + formattedDate); // Output: Formatted Date: 27-04-2024

}

}

**28. Best Practices**

**28.1. Naming Conventions**

* **Classes and Interfaces**: PascalCase (e.g., MyClass, DataProcessor)
* **Methods and Variables**: camelCase (e.g., calculateTotal, userName)
* **Constants**: UPPER\_SNAKE\_CASE (e.g., MAX\_VALUE)

**28.2. Code Readability**

* Use meaningful variable and method names.
* Keep methods short and focused on a single task.
* Use indentation and whitespace effectively.
* Comment complex logic where necessary.

**28.3. Error Handling**

* Handle exceptions gracefully without exposing sensitive information.
* Use specific exceptions rather than generic ones.
* Clean up resources in finally blocks or use try-with-resources.

**28.4. Avoid Premature Optimization**

* Write clear and correct code first.
* Optimize only when necessary based on profiling.

**28.5. Use Java Standard Libraries**

* Leverage existing libraries to avoid reinventing the wheel.
* Familiarize yourself with the Collections Framework, Streams API, etc.

**28.6. Practice Object-Oriented Principles**

* Encapsulate data.
* Favor composition over inheritance.
* Use interfaces to define contracts.

**28.7. Write Unit Tests**

* Ensure your code works as expected.
* Use frameworks like JUnit for testing.

**29. Useful Shortcuts and Tips**

**29.1. String Formatting**

Use String.format for formatted strings.

java

Copy code

int age = 25;

String name = "John";

String formatted = String.format("Name: %s, Age: %d", name, age);

System.out.println(formatted); // Output: Name: John, Age: 25

**29.2. Ternary Operator**

Simplify simple if-else statements.

java

Copy code

int a = 10, b = 20;

int max = (a > b) ? a : b;

System.out.println("Max: " + max); // Output: Max: 20

**29.3. Enhanced For Loop**

Simplify iteration over collections and arrays.

java

Copy code

String[] fruits = {"Apple", "Banana", "Cherry"};

for(String fruit : fruits) {

System.out.println(fruit);

}

**29.4. Varargs**

Allow methods to accept variable number of arguments.

java

Copy code

public static void printNumbers(int... numbers) {

for(int num : numbers) {

System.out.println(num);

}

}

public static void main(String[] args) {

printNumbers(1, 2, 3, 4, 5);

}

**29.5. AutoBoxing and Unboxing**

Automatic conversion between primitive types and their corresponding object wrapper classes.

java

Copy code

Integer obj = 10; // AutoBoxing

int num = obj; // Unboxing

**30. Additional Topics**

**30.1. Reflection**

Inspect and manipulate classes at runtime.

java

Copy code

import java.lang.reflect.Method;

public class ReflectionExample {

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

Class<?> clazz = Class.forName("java.util.ArrayList");

Method[] methods = clazz.getDeclaredMethods();

for(Method method : methods) {

System.out.println(method.getName());

}

}

}

**30.2. Annotations Processing**

Custom processing of annotations during compile-time or runtime.

**30.3. Java Modules (Java 9+)**

Encapsulate packages and control access using modules.

java

Copy code

// module-info.java

module com.example.myapp {

requires java.base;

exports com.example.myapp.utils;

}

**30.4. Functional Interfaces**

Interfaces with a single abstract method, used in Lambda expressions.

java

Copy code

@FunctionalInterface

interface Converter<F, T> {

T convert(F from);

}

public class FunctionalInterfaceExample {

public static void main(String[] args) {

Converter<String, Integer> stringToInteger = Integer::valueOf;

Integer num = stringToInteger.convert("123");

System.out.println(num); // Output: 123

}

}

**31. Helpful Tools and IDEs**

* **Eclipse**: Popular open-source IDE with extensive plugin support.
* **IntelliJ IDEA**: Feature-rich IDE with intelligent code assistance.
* **NetBeans**: Open-source IDE with strong support for Java.
* **Maven/Gradle**: Build automation tools.
* **Git**: Version control system.
* **JUnit/TestNG**: Testing frameworks.