**JAVA**

**Day 1**

1. **What is Java?**

Java is a high-level, object-oriented programming language developed by James Gosling at Sun Microsystems. The language was officially released by Sun Microsystems in 1995.

Java is known for its "write once, run anywhere" philosophy.

It is used in a variety of applications, including web development, mobile app development (Android applications are primarily written in Java), enterprise software, and more.’

1. **Features of Java**

1. Simple- Java is very easy to learn, and its syntax is simple, clean and easy to understand.

2. Object-oriented- Java is an object-oriented programming language. We cannot run a java program without creating a class. In java by using OOPS concepts we can achieve readability, security and reusability.

3. Platform Independent- Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere.

4. Secure- Java is best known for its security. Java provides various security features to the programmers to prevent their application from unauthorized access, threats and viruses.

5. Robust- The Java Programming language is robust, which means it is capable of handling unexpected termination of a program. There are 2 reasons behind this, first, it has a most important and helpful feature called Exception Handling. If an exception occurs in java code then no harm will happen whereas, in other low-level languages, the program will crash. HT EASY LEARNING Another reason why Java is strong lies in its memory management features.

6. Interpreted- In programming languages, you have learned that they use either the compiler or an interpreter, but Java programming language uses both a compiler and an interpreter. Java programs are compiled to generate bytecode files then JVM interprets the bytecode file during execution. Along with this JVM also uses a JIT compiler (it increases the speed of execution).

1. **What is JDK, JRE and JVM?**
2. **JVM (Java Virtual Machine)**:
   * Think of the JVM as a machine that can run Java programs. When you write Java code, the JVM is responsible for running it. It's like an interpreter that takes the code you've written and makes your computer understand it.
   * JVM is platform-independent, meaning it can run on any operating system (Windows, Mac, Linux) as long as there's a JVM for that OS.
3. **JRE (Java Runtime Environment)**:
   * The JRE is like a package that contains the JVM and some other tools needed to run Java programs. It provides the environment in which the Java program can run.
   * If you just want to run Java applications, you only need the JRE installed on your computer.
4. **JDK (Java Development Kit)**:
   * The JDK is like a toolbox for Java developers. It includes the JRE (which contains the JVM) and additional tools like a compiler (which turns your Java code into a form that the JVM can understand) and a debugger (which helps you find and fix errors in your code).
   * If you want to write, compile, and run Java programs, you'll need the JDK.

**Java Variables**

Java Variable is a container that holds a value. Each variable has a data type that determines what kind of data it can hold, such as integers, floating-point numbers, characters, or objects.

**1. Local Variables**

* **What are they?**
  + Local variables are declared inside a method, constructor, or block.
  + They are only accessible within the method, constructor, or block where they are declared.
  + Once the method or block ends, the local variable is destroyed, and its value is lost.
* **Example:**

public class Example {

public void greet() {

String message = "Hello, Onkar!"; // Local variable

System.out.println(message);

}

}

* **Explanation:**
  + In this example, message is a local variable because it is declared inside the greet() method. You can't use message outside this method.

**2. Instance Variables**

* **What are they?**
  + Instance variables are declared inside a class but outside any method, constructor, or block.
  + They are called "instance" variables because they are tied to a specific object (or instance) of the class.
  + Each object of the class has its own copy of the instance variable.
* **Example:**

public class Student {

String name; // Instance variable

int age; // Instance variable

public void setDetails(String n, int a) {

name = n;

age = a;

}

public void showDetails() {

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

System.out.println("Age: " + age);

}

}

public class Main {

public static void main(String[] args) {

Student student1 = new Student();

student1.setDetails("Onkar", 21);

Student student2 = new Student();

student2.setDetails("John", 22);

student1.showDetails(); // Outputs: Name: Onkar, Age: 21

student2.showDetails(); // Outputs: Name: John, Age: 22

}

}

* **Explanation:**
  + name and age are instance variables because they belong to an instance of the Student class. Each Student object (student1 and student2) has its own name and age.

**3. Static Variables**

* **What are they?**
  + A **static variable** belongs to the **class** rather than to any specific object created from the class.
  + This means that **all objects** of that class share the same static variable. If one object changes the value of the static variable, the change is reflected across all other objects.
  + It is declared with the keyword static.All objects of the class share the same static variable.
* **Example:**

public class Counter {

static int count = 0; // Static variable

public Counter() {

count++; // Increment the static variable

}

public void showCount() {

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

}

}

public class Main {

public static void main(String[] args) {

Counter counter1 = new Counter();

Counter counter2 = new Counter();

Counter counter3 = new Counter();

counter3.showCount(); // Outputs: Count: 3

}

}

* **Explanation:**
  + count is a static variable because it's shared by all objects of the Counter class. When counter1, counter2, and counter3 are created, count is incremented each time, and the final value is shared by all objects.

Types of Operators:

▪ Unary Operator (‘-‘, ‘++’, ‘--‘, ‘!’)

▪ Arithmetic operator (+,- ,/,\*,%)

▪ Logical Operator (&& (AND), || (OR), ! (NOT))

▪ Relational Operator (==, !=, >, >=, < , <= )

▪ Assignment Operator (= : Simple assignment • += : Add and assign • -= : Subtract and assign • \*= : Multiply and assign • /= : Divide and assign • %= : Modulus and assign )

▪ Bitwise Operator (& : Bitwise AND • | : Bitwise OR • ^ : Bitwise XOR • ~ : Bitwise complement • << : Left shift • >> : Right shift • >>>Unsigned right shift)

▪ Ternary operator (Statement ? true : false ;)

**First Program : =**

//Save file with same file name as DisplayHelloWorld.java

public class DisplayHelloWorld {

public static void main(String args[]){

System.out.println(“Hello World”);

}

}

Run File in CMD :

1. Navigate to where file is save
2. Use command ….. file\_path\_$$>> **javac file\_name.java**
3. File\_path\_$$>> **java file\_name**

Details of Program :-

**1. public:**

* **Meaning:** public is an access modifier. It means that the class or method is accessible from anywhere in your program.
* **In this code:** The DisplayHelloWorld class and the main method are both public, meaning they can be accessed from outside the class.

**2. class:**

* **Meaning:** class is a keyword used to define a class in Java. A class is like a blueprint for creating objects. It can contain methods (functions) and variables (data).
* **In this code:** DisplayHelloWorld is the name of the class. It defines the structure of what the program will do.

**3. DisplayHelloWorld:**

* **Meaning:** This is the name of the class. You can name a class anything you want, following Java naming conventions.
* **In this code:** DisplayHelloWorld is the name given to the class. It's a custom name chosen to represent what the program does (displaying "Hello World").

**4. { }:**

* **Meaning:** Curly braces { } define a block of code. Everything inside these braces belongs to the class or method they are part of.

**5. public static void main(String args[]):**

* **public:** The main method is public, meaning it can be called by any code outside the class.
* **static:** static means that this method belongs to the class, not to any specific object created from the class. It can be run without creating an object of the class.
* **void:** This means that the method does not return any value. The main method just runs and doesn’t give any result back.
* **main:** This is the name of the method. The main method is special because it is the entry point of any Java program. When you run the program, this method is executed first.
* **String args[]:** This is a way to pass data to the program when you run it. args is an array of String (text) values. In this example, args isn’t used, but it’s a standard part of the main method.

**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 static variable within the System class. It represents the output stream, which is usually the console (the screen where text is displayed).
* **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.

**What happens at runtime?**

 **Class Loading**:

* When you run a Java program, the Java Virtual Machine (JVM) starts by loading the necessary classes. The class loader is responsible for finding and loading class files (bytecode) into memory. It loads classes as needed, starting with the entry point, typically the main method.

 **Bytecode Verification**:

* After the class is loaded, the JVM performs a verification process on the bytecode to ensure it adheres to Java's security constraints. This step prevents malicious or erroneous code from causing harm to the system.

 **Just-In-Time (JIT) Compilation**:

* Java programs are compiled into bytecode, which is platform-independent. During runtime, the JVM uses the Just-In-Time (JIT) compiler to convert the bytecode into machine code specific to the host system. This conversion happens just before the code is executed, which helps optimize performance.

 **Execution of Bytecode**:

* Once the bytecode is compiled to machine code, the JVM's execution engine executes the code. The execution engine interprets or compiles the bytecode into instructions that the computer’s CPU can understand.

 **Memory Management**:

* The JVM manages memory through an automatic process known as garbage collection. It tracks object creation and identifies objects that are no longer in use, freeing up memory resources. This helps prevent memory leaks and ensures efficient memory use.

 **Thread Management**:

* Java supports multithreading, allowing multiple threads to run concurrently within a program. The JVM manages the lifecycle of threads, handling their creation, execution, synchronization, and termination.

 **Exception Handling**:

* During execution, if an unexpected event or error occurs, Java’s built-in exception handling mechanism comes into play. The JVM looks for an appropriate exception handler to manage the error, ensuring the program doesn’t crash unexpectedly.

 **Dynamic Linking**:

* The JVM dynamically links classes and methods as needed. This means that methods are resolved at runtime, allowing for more flexibility and enabling features like polymorphism and dynamic method invocation.

**DAY 2**

**Control Flow (if, switch, loops) :-**

**If Statements**

An if statement lets your program make decisions. You can use it to run a specific block of code only when a certain condition is true.

* **Basic Example**:

**int age = 18;**

**if (age >= 18) {**

**System.out.println("You are an adult.");**

**}**

* + **Explanation**: If age is 18 or more, the program will print "You are an adult." If it's less, nothing happens.
* **If-Else**:

**int age = 16;**

**if (age >= 18) {**

**System.out.println("You are an adult.");**

**} else {**

**System.out.println("You are not an adult.");**

**}**

* + **Explanation**: If age is 18 or more, it prints "You are an adult." If it's less, it prints "You are not an adult."
* **If-Else If-Else**:

int age = 16;

if (age >= 18) {

System.out.println("You are an adult.");

} else if (age >= 13) {

System.out.println("You are a teenager.");

} else {

System.out.println("You are a child.");

}

* + **Explanation**: The program checks multiple conditions. If age is 18 or more, it prints "You are an adult." If it's 13 to 17, it prints "You are a teenager." If it's less than 13, it prints "You are a child."

**2. Switch Statements**

A switch statement is used when you have many possible conditions to check and you want to simplify your code. It works well with a variable that can have multiple values.

* **Basic Example**:

int day = 3;

switch (day) {

case 1:

System.out.println("Monday");

break;

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday");

break;

default:

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

}

* + **Explanation**: The value of day is checked against each case. Since day is 3, it prints "Wednesday." The break statement stops the program from checking further cases. The default case runs if no other cases match.

**3. Loops**

Loops allow your program to repeat a block of code multiple times. This is useful when you want to do something repeatedly, like counting numbers or processing items in a list.

* **For Loop**:
  + **Example**:

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

System.out.println(i);

}

* + - **Explanation**: This loop starts with i as 0 and runs as long as i is less than 5. It increases i by 1 after each loop, printing the numbers 0 to 4.
* **While Loop**:
  + **Example**:

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

* + - **Explanation**: This loop does the same thing as the for loop above, but it's written differently. It keeps running as long as i is less than 5.
* **Do-While Loop**:
  + **Example**:

int i = 0;

do {

System.out.println(i);

i++;

} while (i < 5);

* + - **Explanation**: This loop is similar to a while loop, but it guarantees that the code inside will run at least once, even if the condition is false from the start.

**Functions and Methods**

**1. Functions**

* **General Concept**: A function is a block of code designed to perform a specific task. It can accept input, process it, and return a result. Functions help in reusing code and breaking down complex problems into smaller, manageable parts.
* **Standalone Functions**: In many programming languages, functions exist independently, outside any class. For example:
  + **Python**:

**def add(a, b):**

**return a + b**

Here, add is a standalone function that takes two parameters, a and b, and returns their sum. It can be called anywhere in the code without the need for a class.

* **In Java**: Java does not support standalone functions. All functions in Java must be defined within a class, and hence, they are called **methods**.

**2. Methods**

In Java, a method is similar to a function, but it must be a part of a class. Methods define the behavior of objects in object-oriented programming (OOP). Let’s explore methods in detail:

**Types of Methods in Java**

1. **Instance Methods**:
   * These methods belong to an instance of a class (an object).
   * They can access and modify instance variables (fields) of the class.
   * **Example**:

**class Calculator {**

**int result;**

**// Instance method**

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

**result = a + b;**

**return result;**

**}**

**}**

**public class Main {**

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

**Calculator calc = new Calculator(); // Creating an object**

**int sum = calc.add(5, 3); // Calling instance method**

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

**}**

**}**

* + - **Explanation**: Here, add is an instance method. It operates on the instance (object) calc of the Calculator class and can access the instance variable result.

1. **Static Methods**:
   * These methods belong to the class itself rather than any object.
   * They can be called without creating an instance of the class.
   * Static methods can only directly access other static members (variables or methods) of the class.
   * **Example**:

**class MathUtil {**

**// Static method**

**public static int multiply(int a, int b) {**

**return a \* b;**

**}**

**}**

**public class Main {**

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

**int product = MathUtil.multiply(5, 3); // Calling static method**

**System.out.println("Product: " + product); // Output: Product: 15**

**}**

**}**

* + - **Explanation**: multiply is a static method that can be called directly using the class name MathUtil without creating an instance.

1. **Void vs. Return Methods**:
   * **Void Methods**: Do not return a value. They perform an action but don’t send any result back.

**class Printer {**

**// Void method**

**public void printMessage() {**

**System.out.println("Hello, World!");**

**}**

**}**

* + **Return Methods**: These methods return a value after execution.

**class Calculator {**

**public int square(int number) {**

**return number \* number;**

**}**

**}**

**Method Overloading and Overriding**

1. **Overloading**:
   * Allows you to define multiple methods with the same name but different parameter lists (different types or numbers of parameters).
   * **Example**:

**class Calculator {**

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

**return a + b;**

**}**

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

**return a + b;**

**}**

**}**

* + - **Explanation**: The add method is overloaded to handle both int and double parameters.

1. **Overriding**:
   * Allows a subclass to provide a specific implementation of a method that is already defined in its superclass.
   * **Example**:

**class Animal {**

**public void sound() {**

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

**}**

**}**

**class Dog extends Animal {**

**@Override**

**public void sound() {**

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

**}**

**}**

* + - **Explanation**: The Dog class overrides the sound method of the Animal class to provide a specific implementation.

**3. Key Differences Between Functions and Methods**

1. **Existence in Java**:
   * **Functions**: In general, a function can exist independently of any class (in other languages like C or Python).
   * **Methods**: In Java, every "function" is a method, meaning it must be associated with a class.
2. **Association with Classes**:
   * **Functions**: Standalone functions are not tied to any class or object.
   * **Methods**: Methods in Java are always tied to a class or an object.
3. **Invocation**:
   * **Functions**: Standalone functions are called directly by their name in languages that support them.
   * **Methods**: Methods are called using the object or class they belong to. Instance methods require an object, while static methods can be called directly using the class name.
4. **Access to Data**:
   * **Functions**: In languages that support standalone functions, they do not have direct access to the internal data of objects.
   * **Methods**: In Java, methods can access and modify the internal state (fields) of the objects they belong to, making them powerful in the context of OOP.
5. **Overloading and Overriding**:
   * **Functions**: Can be overloaded in some languages, but do not support overriding (since they are not tied to classes).
   * **Methods**: In Java, methods can be both overloaded (same name, different parameters) and overridden (in subclasses).

**Day 3**

**Object-Oriented Programming o Classes and Objects o Inheritance and Polymorphism o Encapsulation and Abstraction**

**Class**

A **class** is a blueprint or template that defines the structure and behavior of objects. It encapsulates data (attributes or fields) and methods (functions or behaviors) that operate on the data. A class is a user-defined data type that serves as a prototype for creating objects.

**Object**

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object is created from it. Objects are the actual entities that interact with one another in a program. Each object has its own set of attributes and can perform actions defined by the methods in its class.

public class Car {

// Attributes (fields)

String color;

String model;

int speed;

// Constructor

public Car(String color, String model, int speed) {

this.color = color;

this.model = model;

this.speed = speed;

}

// Method

public void accelerate(int increment) {

speed += increment;

}

public void brake(int decrement) {

speed -= decrement;

}

}

public class Main {

public static void main(String[] args) {

// Creating an object of the Car class

Car myCar = new Car("Red", "Toyota", 0);

// Accessing the object's attributes and methods

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

System.out.println("Initial Speed: " + myCar.speed);

// Using the accelerate method

myCar.accelerate(50);

System.out.println("Speed after accelerating: " + myCar.speed);

// Using the brake method

myCar.brake(20);

System.out.println("Speed after braking: " + myCar.speed);

}

}

**1. Inheritance**

**Inheritance** is a mechanism that allows a new class to inherit the properties and behaviors (fields and methods) of an existing class. The existing class is called the **superclass** or **parent class**, and the new class is called the **subclass** or **child class**.

**Key Points:**

* **Reusability**: Inheritance promotes code reusability by allowing the child class to reuse code from the parent class.
* **Hierarchy**: It creates a hierarchical relationship between classes.

// Superclass (Parent Class)

class Animal {

String name;

public void eat() {

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

}

}

// Subclass (Child Class)

class Dog extends Animal {

public void bark() {

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

}

}

public class Main {

public static void main(String[] args) {

// Create an object of the Dog class

Dog myDog = new Dog();

myDog.name = "Buddy";

// Access methods from both the superclass and subclass

myDog.eat(); // Inherited from Animal class

myDog.bark(); // Defined in Dog class

}

}

**2.Polymorphism**

**Polymorphism** allows objects to be treated as instances of their parent class, enabling multiple implementations of the same method.

**Polymorphism** means "many forms." It allows objects to be treated as instances of their parent class, even if they are instances of different child classes. Polymorphism can be achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism).

**I. Method Overloading (Compile-time Polymorphism)**

**Method Overloading** occurs when multiple methods in the same class share the same name but have different parameters (different number, type, or both). The method that gets called is determined at compile time, hence the name compile-time polymorphism.

class Calculator {

// Method to add two integers

int add(int a, int b) {

return a + b;

}

// Overloaded method to add three integers

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

return a + c + b;

}

// Overloaded method to add two double values

double add(double a, double b) {

return a + b;

}

}

public class Main {

public static void main(String[] args) {

Calculator calc = new Calculator();

// Calling different overloaded methods

System.out.println(calc.add(5, 10)); // Calls add(int, int)

System.out.println(calc.add(5, 10, 15)); // Calls add(int, int, int)

System.out.println(calc.add(5.5, 4.5)); // Calls add(double, double)

}

}

II. **Method Overriding (Runtime Polymorphism)**

**Method Overriding** happens when a subclass (child class) provides a specific implementation of a method that is already defined in its superclass (parent class). The method to be executed is determined at runtime, hence the name runtime polymorphism.

// Parent class

class Animal {

// Method to be overridden

public void sound() {

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

}

}

// Child class

class Dog extends Animal {

// Overriding the sound method

@Override

public void sound() {

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

}

}

public class Main {

public static void main(String[] args) {

Animal myAnimal = new Animal(); // Parent class reference and object

Animal myDog = new Dog(); // Parent class reference but Dog object

// Calling the sound method

myAnimal.sound(); // Calls Animal's sound method

myDog.sound(); // Calls Dog's overridden sound method

}

}

**3.Abstraction**

**Abstraction** in Java is one of the key principles of Object-Oriented Programming (OOP). It focuses on hiding the complex implementation details and exposing only the essential features or functionality to the user. Abstraction allows you to work with high-level concepts without needing to understand the low-level details.

Imagine you're driving a car. You don't need to know the intricate mechanics of how the engine works; you only need to know how to operate the steering wheel, brakes, and accelerator. Similarly, in programming, abstraction hides the internal workings and exposes only what is necessary to use an object effectively.

// Abstract class

abstract class Vehicle {

// Abstract method (does not have a body)

abstract void start();

// Non-abstract method (has a body)

void stop() {

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

}

}

// Subclass that extends the abstract class

class Car extends Vehicle {

// Providing implementation for the abstract method

@Override

void start() {

System.out.println("Car is starting");

}

}

public class Main {

public static void main(String[] args) {

// Vehicle myVehicle = new Vehicle(); // Error! Cannot instantiate abstract class

Vehicle myCar = new Car(); // Creating an object of the subclass

myCar.start(); // Calls the implemented start method in Car class

myCar.stop(); // Calls the non-abstract stop method from Vehicle class

}

}

**4. Encapsulation**

**Encapsulation** is the technique of wrapping data (attributes) and methods (functions) into a single unit, typically a class. It also involves restricting direct access to some of an object’s components, which is achieved using access modifiers (private, protected, public).

class Employee {

// Private variables (encapsulated fields)

private String name;

private int age;

// Getter method for name

public String getName() {

return name;

}

// Setter method for name

public void setName(String name) {

this.name = name;

}

// Getter method for age

public int getAge() {

return age;

}

// Setter method for age

public void setAge(int age) {

if (age > 0) { // Adding a condition for validation

this.age = age;

} else {

System.out.println("Age must be positive");

}

}

}

public class Main {

public static void main(String[] args) {

Employee emp = new Employee();

// Setting values using setter methods

emp.setName("John Doe");

emp.setAge(30);

// Accessing values using getter methods

System.out.println("Employee Name: " + emp.getName());

System.out.println("Employee Age: " + emp.getAge());

}

}

**Java Packages**

A **package** in Java is like a folder in your computer that helps you organize your files. Just as you store related documents in the same folder, in Java, you use packages to group related classes, interfaces, and sub-packages. Packages help you manage your code better, avoid naming conflicts, and control access to your classes and methods.

**Why Use Packages?**

1. **Organization**: Packages group related classes and interfaces together. For example, if you're working on a project with multiple modules (like user management, product management, etc.), you can create a separate package for each module.
2. **Avoiding Name Conflicts**: In a large project, different parts of the project might have classes with the same name. Packages help avoid conflicts by providing a unique namespace for each class.
3. **Access Control**: Packages can restrict access to certain classes, methods, and variables. This means you can control which parts of your code are accessible from outside the package.
4. **Code Reusability**: By organizing your code into packages, you can easily reuse classes and interfaces in other projects.

**Types of Packages**

Java has two main types of packages:

1. **Built-in Packages**: These are packages that come with the Java Standard Library. For example:
   * java.lang: Contains core classes like String, System, Math, etc.
   * java.util: Contains utility classes like ArrayList, HashMap, Date, etc.
   * java.io: Contains classes for input/output operations like File, InputStream, OutputStream, etc.
2. **User-defined Packages**: These are packages that you create to organize your own classes and interfaces.

**Creating a Package**

To create a package, you simply use the package keyword at the beginning of your Java file, followed by the package name. The package name is usually written in lowercase to avoid conflicts with class names.

**Example of Creating a Package:**

Let's say you're working on a project related to a library system, and you want to create a package to manage books.

// File: Book.java

package library.books; // Declaring the package

public class Book {

private String title;

private String author;

public Book(String title, String author) {

this.title = title;

this.author = author;

}

public void displayInfo() {

System.out.println("Title: " + title);

System.out.println("Author: " + author);

}

}

Here, library.books is the package name, and the Book class is part of this package.

**Using a Package**

Once you've created a package, you can use the classes in that package in other classes by importing the package.

**Example of Using a Package:**

// File: LibraryMain.java

package library.main;

import library.books.Book; // Importing the Book class from library.books package

public class LibraryMain {

public static void main(String[] args) {

Book myBook = new Book("1984", "George Orwell");

myBook.displayInfo();

}

}

In this example, the LibraryMain class is in the library.main package, and it imports the Book class from the library.books package.

**Package Naming Conventions**

Package names are usually written in all lowercase letters. If your project is large and has many packages, you might use a naming convention based on your domain name. For example, if your domain is example.com, you might name your packages like this:

* com.example.library.books
* com.example.library.members
* com.example.library.transactions

This helps in organizing and avoiding conflicts with other packages that might have similar names.

**Access Modifiers and Packages**

Java has four access modifiers: public, protected, default (no modifier), and private. These modifiers control access to classes and their members (fields, methods, constructors) from other classes.

* **public**: The class or member is accessible from any other class.
* **protected**: The class or member is accessible within its own package and by subclasses.
* **default** (no modifier): The class or member is accessible only within its own package.
* **private**: The class or member is accessible only within its own class.

**Access Modifiers in Java**

Access modifiers in Java are keywords that set the accessibility or visibility of classes, methods, and other members. They determine whether other classes can use a particular field, method, constructor, or class.

Java has four main access modifiers:

1. **public**
2. **protected**
3. **default** (no modifier)
4. **private**

**1. public**

* **What it means**: The public access modifier allows a class, method, or variable to be accessible from any other class in any package.
* **Usage**: If a member is declared as public, it can be accessed from any other class, regardless of the package in which the accessing class is located.

**Example:**

public class Animal {

public String name;

public void sound() {

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

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Animal();

animal.name = "Dog"; // Accessible because name is public

animal.sound(); // Accessible because sound() is public

}

}

**2. protected**

* **What it means**: The protected access modifier allows the member to be accessible within its own package and by subclasses in other packages.
* **Usage**: protected is often used when you want to give access to classes in the same package and to subclasses, but not to other unrelated classes.

**Example:**

package animals;

public class Animal {

protected String name;

protected void sound() {

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

}

}

package main;

import animals.Animal;

public class Dog extends Animal {

public void display() {

name = "Dog"; // Accessible because Dog is a subclass

sound(); // Accessible because Dog is a subclass

}

}

**3. default (no modifier)**

* **What it means**: When no access modifier is specified, Java applies the default access level, also known as package-private. It allows access only within the same package.
* **Usage**: default access is useful when you want to keep members accessible only to other classes in the same package but hidden from classes in other packages.

**Example:**

class Animal {

String name; // Default access

void sound() { // Default access

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

}

}

class Main {

public static void main(String[] args) {

Animal animal = new Animal();

animal.name = "Dog"; // Accessible because Main is in the same package

animal.sound(); // Accessible because Main is in the same package

}

}

**4. private**

* **What it means**: The private access modifier restricts access to the member so that it is only accessible within the class it is declared in.
* **Usage**: private is used when you want to completely hide the member from other classes, even within the same package.

**Example:**

public class Animal {

private String name;

private void sound() {

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

}

public void setName(String name) {

this.name = name;

}

public void displaySound() {

sound(); // sound() is accessible within the same class

}

}

public class Main {

public static void main(String[] args) {

Animal animal = new Animal();

animal.setName("Dog"); // Accessible because setName() is public

animal.displaySound(); // Accessible because displaySound() is public

// animal.name = "Cat"; // Error! name is private

// animal.sound(); // Error! sound() is private

}

}

**Exception Handling in Java**

**Exception Handling** is a mechanism in Java to handle runtime errors, ensuring the normal flow of the application. When an error occurs during the execution of a program, an exception is thrown.

**What is an Exception?**

An **exception** is an event that disrupts the normal flow of the program's instructions.

There are two main types of exceptions in Java:

1. **Checked Exceptions**: These are exceptions that are checked at compile-time. The compiler forces the programmer to handle these exceptions. Examples include IOException, SQLException, etc.
2. **Unchecked Exceptions**: These are exceptions that occur at runtime and are not checked by the compiler. Examples include ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc.

**Exception Handling Mechanism**

Java provides five key keywords for exception handling:

1. **try**
2. **catch**
3. **finally**
4. **throw**
5. **throws**

**1. Try Block**

The try block is used to enclose code that might throw an exception. If an exception occurs, it is caught by the catch block. A try block must be followed by one or more catch blocks or a finally block.

try {

// Code that may throw an exception

int result = 10 / 0;

}

**2. catch Block**

The catch block is used to handle the exception that occurs in the try block. You can have multiple catch blocks to handle different types of exceptions.

catch (ArithmeticException e) {

// Code to handle the exception

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

}

**3. finally Block**

The finally block contains code that is always executed, whether an exception is thrown or not. It is typically used to release resources like closing a file or a database connection.

finally {

// Code that will always execute

System.out.println("This is the finally block");

}

**4. throw Keyword**

The throw keyword is used to explicitly throw an exception. This is useful when you want to create your own custom exceptions or handle specific cases.

throw new ArithmeticException("Cannot divide by zero");

**5. throws Keyword**

The throws keyword is used in a method signature to indicate that the method might throw one or more exceptions. This forces the calling method to handle those exceptions.

public void divide() throws ArithmeticException {

int result = 10 / 0;

}

public class ExceptionExample {

public static void main(String[] args) {

try {

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

System.out.println(numbers[5]); // This will throw ArrayIndexOutOfBoundsException

} catch (ArrayIndexOutOfBoundsException e) {

System.out.println("Array index is out of bounds!");

} catch (Exception e) {

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

} finally {

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

}

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

}

}

**Java Multithreading**

**Multithreading** in Java allows you to run multiple threads (smaller, lightweight processes) concurrently. This means you can perform multiple tasks simultaneously within a single program, making your application more efficient and responsive.

**What is a Thread?**

A **thread** is the smallest unit of execution in a program. In Java, every application has at least one thread (the main thread) that runs the main() method. By creating additional threads, you can perform multiple operations at the same time.

**Why Use Multithreading?**

* **Improved Performance**: By running tasks in parallel, multithreading can make your program faster, especially on multi-core processors.
* **Responsiveness**: Multithreading allows your application to remain responsive. For example, a user interface can stay active while performing background tasks.
* **Resource Sharing**: Threads can share resources like memory, which can reduce the overhead of creating multiple processes.

**How to Create Threads in Java**

There are two main ways to create a thread in Java:

1. **By Extending the Thread Class**
2. **By Implementing the Runnable Interface**

**1. Extending the Thread Class**

When you extend the Thread class, you override its run() method, which contains the code you want the thread to execute.

**Example:**

class MyThread extends Thread {

public void run() {

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

System.out.println(Thread.currentThread().getName() + " - " + i);

}

}

}

public class Main {

public static void main(String[] args) {

MyThread t1 = new MyThread();

MyThread t2 = new MyThread();

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

t2.start(); // Starts the second thread

}

}

**Output:**

Thread-0 - 0

Thread-1 - 0

Thread-0 - 1

Thread-1 - 1

...

Here, t1.start() and t2.start() start two threads that run concurrently, each executing the run() method. The output may interleave because both threads are running at the same time.

**2. Implementing the Runnable Interface**

The Runnable interface is a functional interface with a single method run(). You implement this interface when your class needs to perform work in a thread.

**Example:**

class MyRunnable implements Runnable {

public void run() {

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

System.out.println(Thread.currentThread().getName() + " - " + i);

}

}

}

public class Main {

public static void main(String[] args) {

MyRunnable runnable = new MyRunnable();

Thread t1 = new Thread(runnable);

Thread t2 = new Thread(runnable);

t1.start();

t2.start();

}

}

**Output:**

mathematica

Copy code

Thread-0 - 0

Thread-1 - 0

Thread-0 - 1

Thread-1 - 1

...

**Understanding Thread Lifecycle**

A thread in Java can be in one of the following states:

1. **New**: When a thread is created but not yet started.
2. **Runnable**: When the thread is ready to run and waiting for CPU time.
3. **Running**: When the thread is executing its task.
4. **Blocked**: When the thread is waiting for a resource (like I/O or a lock).
5. **Terminated**: When the thread has finished executing.

**Day 4**

**Hash Tables**

**Hash tables are data structures that store key-value pairs. They are efficient for lookup, insertion, and deletion operations. The key is passed through a hash function, which generates an index for storing the associated value in the table. The goal is to achieve constant time complexity, O(1), for these operations.**

**Hash Functions**

**A hash function takes an input (or 'key') and returns an integer, which is used as an index in a hash table. The quality of a hash function is crucial for distributing the keys uniformly across the hash table, minimizing collisions.**

**Properties of a good hash function:**

1. **Deterministic: The same key should always produce the same hash value.**
2. **Uniformity: The hash function should distribute keys uniformly across the hash table to avoid clustering.**
3. **Fast computation: The hash function should be quick to compute.**
4. **Minimal collisions: It should minimize the number of collisions, where two different keys produce the same hash value.**

**Example: If you have a hash table of size 10, and your key is "apple", a simple hash function might convert the string into an integer (using the ASCII values of the characters) and then take the remainder when divided by 10.**

**Handling Collisions**

**Collisions occur when two different keys produce the same hash value and are assigned the same index in the hash table. There are several strategies to handle collisions:**

1. **Chaining:**
   * **In chaining, each index in the hash table points to a linked list (or another data structure) containing all the elements that hash to the same index.**
   * **When a collision occurs, the new key-value pair is added to the list at the index.**
   * **Advantages: Simple and can handle a large number of collisions.**
   * **Disadvantages: Requires additional memory for pointers, and the worst-case lookup time is O(n) when all keys collide.**

**Example:**

**Index 0: None**

**Index 1: (apple, 5) -> (banana, 7)**

**Index 2: None**

1. **Open Addressing:**
   * **In open addressing, all elements are stored in the hash table itself. When a collision occurs, the algorithm searches for the next available slot in the table using a probing technique.**
   * **Probing Techniques:**
     + **Linear Probing: Check the next slot (index + 1) until an empty slot is found.**
     + **Quadratic Probing: Check slots at intervals of 1, 4, 9, etc. (i.e., index + 1^2, index + 2^2, etc.)**
     + **Double Hashing: Use a second hash function to calculate the step size for probing.**
   * **Advantages: Efficient memory usage as all elements are stored in the hash table.**
   * **Disadvantages: Can lead to clustering, where a group of close slots are occupied, reducing efficiency.**

**Example (Linear Probing):**

**Index 0: (apple, 5)**

**Index 1: (banana, 7)**

**Index 2: None (If "cherry" hashes to index 1, it will be placed here)**

**Design an LRU (Least Recently Used) Cache**

**1. Concept**

* **An LRU Cache is a data structure that stores a fixed number of items while maintaining access order.**
* **When the cache reaches its capacity, the least recently used item is removed to make room for new items.**

**2. Key Operations**

* **Get (key): Retrieve the value associated with the key. If the key is not present, return -1.**
* **Put (key, value): Insert or update the key-value pair in the cache. If the cache exceeds its capacity, evict the least recently used item.**

**3. Data Structures**

* **Doubly Linked List:**
  + **Stores key-value pairs in nodes.**
  + **Allows quick addition and removal of nodes from both ends.**
  + **The most recently used item is at the head, and the least recently used item is at the tail.**
* **HashMap:**
  + **Maps keys to their corresponding nodes in the doubly linked list.**
  + **Provides O(1) time complexity for both the get and put operations.**

**4. Algorithm**

* **Get Operation:**
  1. **Check if the key exists in the HashMap.**
  2. **If the key exists, move the corresponding node to the front of the doubly linked list to mark it as recently used.**
  3. **Return the value associated with the key.**
  4. **If the key does not exist, return -1.**
* **Put Operation:**
  1. **Check if the key already exists in the HashMap.**
  2. **If it exists, update the value and move the node to the front of the doubly linked list.**
  3. **If the key does not exist, create a new node.**
  4. **If the cache is at capacity, remove the node at the tail of the doubly linked list (the least recently used item) and delete its entry in the HashMap.**
  5. **Insert the new node at the front of the doubly linked list and add it to the HashMap.**

**5. Efficiency**

* **Both get and put operations run in O(1) time due to the combination of the doubly linked list and HashMap.**

**import java.util.HashMap;**

**class LRUCache {**

**class Node {**

**int key, value;**

**Node prev, next;**

**Node(int key, int value) {**

**this.key = key;**

**this.value = value;**

**}**

**}**

**private int capacity;**

**private HashMap<Integer, Node> map;**

**private Node head, tail;**

**public LRUCache(int capacity) {**

**this.capacity = capacity;**

**this.map = new HashMap<>();**

**this.head = new Node(0, 0);**

**this.tail = new Node(0, 0);**

**head.next = tail;**

**tail.prev = head;**

**}**

**public int get(int key) {**

**if (map.containsKey(key)) {**

**Node node = map.get(key);**

**remove(node);**

**insertToFront(node);**

**return node.value;**

**} else {**

**return -1;**

**}**

**}**

**public void put(int key, int value) {**

**if (map.containsKey(key)) {**

**remove(map.get(key));**

**}**

**if (map.size() == capacity) {**

**remove(tail.prev);**

**}**

**insertToFront(new Node(key, value));**

**}**

**private void remove(Node node) {**

**map.remove(node.key);**

**node.prev.next = node.next;**

**node.next.prev = node.prev;**

**}**

**private void insertToFront(Node node) {**

**map.put(node.key, node);**

**node.next = head.next;**

**node.prev = head;**

**head.next.prev = node;**

**head.next = node;**

**}**

**}**

**HashMap**

** Two Sum Problem**

* **Description: Given an array of integers, find two numbers such that they add up to a specific target number. Return the indices of the two numbers.**
* **Hint: Use a HashMap to store the difference between the target and the current element as you iterate through the array.**
* ** Input: nums = [2, 7, 11, 15], target = 9**
* ** Expected Output: [0, 1]**
* ** Explanation: The numbers at indices 0 and 1 add up to 9 (2 + 7 = 9).**

** Find the Longest Substring Without Repeating Characters**

* **Description: Given a string, find the length of the longest substring without repeating characters.**
* **Hint: Use a HashMap to store the last seen index of each character as you iterate through the string.**
* ** Input: s = "abcabcbb"**
* ** Expected Output: 3**
* ** Explanation: The longest substring without repeating characters is "abc", which has a length of 3.**

** Count the Number of Subarrays with a Given XOR**

* **Description: Given an array of integers and a target XOR value, find the number of subarrays that have a XOR value equal to the target.**
* **Hint: Use a HashMap to keep track of the XOR of all elements up to the current index.**
* ** Input: arr = [4, 2, 2, 6, 4], target = 6**
* ** Expected Output: 4**
* ** Explanation: The subarrays [4, 2], [2, 6], [6], and [2, 2, 6] have XOR equal to 6.**

** Group Anagrams**

* **Description: Given an array of strings, group the anagrams together.**
* **Hint: Use a HashMap where the key is a sorted version of the string, and the value is a list of strings that are anagrams.**
* ** Input: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]**
* ** Expected Output: [["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]**
* ** Explanation: The anagrams are grouped together.**

** Subarray Sum Equals K**

* **Description: Given an array of integers and an integer k, find the total number of continuous subarrays whose sum equals to k.**
* **Hint: Use a HashMap to store the cumulative sum and its frequency.**
* ** Input: nums = [1, 2, 3], k = 3**
* ** Expected Output: 2**
* ** Explanation: The subarrays [1, 2] and [3] have sums equal to 3.**

** Find All Duplicates in an Array**

* **Description: Given an array of integers, return an array of all the integers that appear twice.**
* **Hint: Use a HashMap to track the frequency of each element.**
* ** Input: nums = [4, 3, 2, 7, 8, 2, 3, 1]**
* ** Expected Output: [2, 3]**
* ** Explanation: The numbers 2 and 3 appear twice in the array.**

** Top K Frequent Elements**

* **Description: Given a non-empty array of integers, return the k most frequent elements.**
* **Hint: Use a HashMap to store the frequency of each element and then sort or use a priority queue to find the top K.**
* ** Input: nums = [1, 1, 1, 2, 2, 3], k = 2**
* ** Expected Output: [1, 2]**
* ** Explanation: The numbers 1 and 2 are the most frequent elements.**

** Intersection of Two Arrays II**

* **Description: Given two arrays, write a function to compute their intersection. Each element in the result should appear as many times as it shows in both arrays.**
* **Hint: Use a HashMap to store the frequency of elements in one array and then iterate through the other array to find common elements.**
*  **Input**: nums1 = [1, 2, 2, 1], nums2 = [2, 2]
* **Expected Output**: [2, 2]
* **Explanation**: The intersection includes two occurrences of 2.
* 

** Copy List with Random Pointer**

* **Description: A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null. Return a deep copy of the list.**
* **Hint: Use a HashMap to store the mapping from the original node to the new node.**
* ** Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]**
* ** Expected Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]**
* ** Explanation: A deep copy of the list with the same structure is created.**

** Word Pattern**

* **Description: Given a pattern and a string, find if the string follows the same pattern.**
* **Hint: Use two HashMaps to maintain a mapping from pattern to string and string to pattern.**
  + **Input**: pattern = "abba", str = "dog cat cat dog"
  + **Expected Output**: true
  + **Explanation**: The string follows the pattern where "a" maps to "dog" and "b" maps to "cat".

4o



**key functions (methods) provided by HashMap**

**1. put(K key, V value)**

* **Description: Inserts a key-value pair into the HashMap. If the key already exists, the old value is replaced with the new value.**
* **Example:**

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

**map.put("Apple", 3);**

**map.put("Banana", 5);**

**map.put("Apple", 4); // Replaces the value associated with "Apple" with 4**

**2. get(Object key)**

* **Description: Retrieves the value associated with the specified key. Returns null if the key is not found.**
* **Example:**

**int appleCount = map.get("Apple"); // Returns 4**

**int orangeCount = map.get("Orange"); // Returns null (key not found)**

**3. containsKey(Object key)**

* **Description: Checks if the HashMap contains the specified key.**
* **Example:**

**if (map.containsKey("Banana")) {**

**System.out.println("Banana is in the map!");**

**}**

**4. containsValue(Object value)**

* **Description: Checks if the HashMap contains the specified value.**
* **Example:**

**if (map.containsValue(4)) {**

**System.out.println("Some fruit has a count of 4!");**

**}**

**5. remove(Object key)**

* **Description: Removes the key-value pair associated with the specified key.**
* **Example:**

**map.remove("Apple"); // Removes the key "Apple" and its associated value**

**6. size()**

* **Description: Returns the number of key-value pairs in the HashMap.**
* **Example:**

**int size = map.size(); // Returns 1 since "Apple" was removed**

**7. isEmpty()**

* **Description: Checks if the HashMap is empty (contains no key-value pairs).**
* **Example:**

**if (map.isEmpty()) {**

**System.out.println("The map is empty!");**

**}**

**8. keySet()**

* **Description: Returns a Set view of the keys contained in the HashMap.**
* **Example:**

**Set<String> keys = map.keySet(); // Returns a set of keys: {"Banana"}**

**9. values()**

* **Description: Returns a Collection view of the values contained in the HashMap.**
* **Example:**

**Collection<Integer> values = map.values(); // Returns a collection of values: {5}**

**10. entrySet()**

**- \*\*Description\*\*: Returns a `Set` view of the key-value pairs contained in the `HashMap`.**

**- \*\*Example\*\*:**

**```java**

**Set<Map.Entry<String, Integer>> entries = map.entrySet();**

**for (Map.Entry<String, Integer> entry : entries) {**

**System.out.println(entry.getKey() + " => " + entry.getValue());**

**}**

**```**

**11. putAll(Map<? extends K, ? extends V> m)**

**- \*\*Description\*\*: Copies all key-value pairs from the specified map to the `HashMap`.**

**- \*\*Example\*\*:**

**```java**

**HashMap<String, Integer> anotherMap = new HashMap<>();**

**anotherMap.put("Orange", 2);**

**anotherMap.put("Grapes", 10);**

**map.putAll(anotherMap); // Adds all entries from anotherMap to map**

**```**

**12. clear()**

**- \*\*Description\*\*: Removes all key-value pairs from the `HashMap`.**

**- \*\*Example\*\*:**

**```java**

**map.clear(); // Now map is empty**

**```**

**13. getOrDefault(Object key, V defaultValue)**

**- \*\*Description\*\*: Returns the value to which the specified key is mapped, or `defaultValue` if the `HashMap` contains no mapping for the key.**

**- \*\*Example\*\*:**

**int appleCount = map.getOrDefault("Apple", 0); // Returns 0 if "Apple" is not found**

**```**

**14. replace(K key, V value)**

**- \*\*Description\*\*: Replaces the entry for the specified key only if it is currently mapped to some value.**

**- \*\*Example\*\*:**

**```java**

**map.replace("Banana", 6); // Replaces the value associated with "Banana" with 6**

**```**

**15. replace(K key, V oldValue, V newValue)**

**- \*\*Description\*\*: Replaces the entry for the specified key only if currently mapped to the specified value.**

**- \*\*Example\*\*:**

**boolean isReplaced = map.replace("Banana", 5, 7); // Only replaces if current value is 5**

**```**

**16. compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**

**- \*\*Description\*\*: Computes a new mapping for the specified key and its current mapped value.**

**- \*\*Example\*\*:**

**map.compute("Banana", (key, val) -> (val == null) ? 1 : val + 1); // Increments the value associated with "Banana" by 1**

**```**

**17. computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)**

**- \*\*Description\*\*: If the specified key is not already associated with a value (or is mapped to `null`), computes its value using the given mapping function and enters it into the map.**

**- \*\*Example\*\*:**

**map.computeIfAbsent("Mango", k -> 1); // Adds "Mango" with value 1 if it's not already in the map**

**```**

**18. computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)**

**- \*\*Description\*\*: If the value for the specified key is present and non-null, computes a new mapping given the key and its current mapped value.**

**- \*\*Example\*\*:**

**map.computeIfPresent("Banana", (key, val) -> val + 1); // Increments the value for "Banana" by 1 if "Banana" exists**

**```**

**19. merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)**

**- \*\*Description\*\*: If the specified key is not already associated with a value or is associated with `null`, associates it with the given value. Otherwise, replaces the value with the result of the given remapping function.**

**- \*\*Example\*\*:**

**map.merge("Banana", 1, Integer::sum); // Adds 1 to the value associated with "Banana" if it exists, otherwise sets it to 1**

**```**

**Real-World Example**

**Let's say you're developing an application that tracks inventory in a store. You can use a HashMap where the key is the product name and the value is the quantity in stock:**

**HashMap<String, Integer> inventory = new HashMap<>();**

**inventory.put("Apple", 50);**

**inventory.put("Banana", 30);**

**// Check if a product is available**

**if (inventory.containsKey("Apple")) {**

**System.out.println("Apples in stock: " + inventory.get("Apple"));**

**}**

**// Update stock**

**inventory.put("Apple", inventory.get("Apple") - 10); // Sold 10 apples**

**// Check if a product is out of stock**

**if (inventory.getOrDefault("Orange", 0) == 0) {**

**System.out.println("Orange is out of stock.");**

**}**

**// Print all products**

**for (Map.Entry<String, Integer> entry : inventory.entrySet()) {**

**System.out.println(entry.getKey() + ": " + entry.getValue() + " units");**

**}**

**Day 5**

**Tree**

**1. Structure of a Binary Tree**

**A binary tree is a hierarchical data structure in which each node has at most two children. These two children are referred to as the left child and the right child.**

* **Node: The basic unit of a binary tree. Each node contains:**
  + **Data: The value stored in the node.**
  + **Left Child: A pointer/reference to another node that represents the left child.**
  + **Right Child: A pointer/reference to another node that represents the right child.**
* **Root Node: The topmost node in a binary tree. It is the starting point of the tree.**
* **Leaf Node: A node that does not have any children (both left and right children are null).**

**Example:**

**A**

**/ \**

**B C**

**/ \ \**

**D E F**

* **A is the root node.**
* **B and C are the children of A.**
* **D, E, and F are the leaf nodes.**

**2. Types of Binary Trees**

**There are several variations of binary trees, each with its own rules and characteristics:**

* **Full Binary Tree: Every node has either 0 or 2 children. There are no nodes with only one child.**
  + **Example:**

**A**

**/ \**

**B C**

**/ \**

**D E**

* **Perfect Binary Tree: All internal nodes have two children, and all leaf nodes are at the same level.**
  + **Example:**

**A**

**/ \**

**B C**

**/ \ / \**

**D E F G**

* **Complete Binary Tree: All levels are fully filled except possibly for the last level, which is filled from left to right.**
  + **Example:**

**A**

**/ \**

**B C**

**/ \ /**

**D E F**

* **Balanced Binary Tree: The difference in height between the left and right subtrees of any node is at most 1. This ensures that the tree remains balanced, making operations like insertion, deletion, and lookup efficient.**
* **Binary Search Tree (BST): A binary tree with an additional constraint: for each node, the left subtree contains values less than the node’s value, and the right subtree contains values greater than the node’s value.**
  + **Example:**

**5**

**/ \**

**3 8**

**/ \ / \**

**2 4 7 9**

**3. Properties of Binary Trees**

* **Height of a Node: The number of edges on the longest path from the node to a leaf. For a leaf node, the height is 0.**
* **Depth of a Node: The number of edges from the root to the node. The root node has a depth of 0.**
* **Level of a Node: The depth of the node plus 1. The root node is at level 1.**
* **Size of a Tree: The total number of nodes in the tree.**

**4. Common Operations on Binary Trees**

* **Traversal: Visiting all the nodes in some order. There are three common types of depth-first traversals:**
  + **In-order Traversal (Left, Root, Right): Visit the left subtree, the root, and then the right subtree.**
  + **Pre-order Traversal (Root, Left, Right): Visit the root, then the left subtree, and finally the right subtree.**
  + **Post-order Traversal (Left, Right, Root): Visit the left subtree, the right subtree, and then the root.**
* **Insertion: Adding a new node to the tree, typically while maintaining the properties of the tree (e.g., in a BST, ensuring the left child is less than the node and the right child is greater).**
* **Deletion: Removing a node from the tree and reorganizing the tree to maintain its properties.**
* **Searching: Finding a node with a given value. In a binary search tree, this can be done efficiently by comparing the value with the root and deciding whether to search in the left or right subtree.**

**5. Applications of Binary Trees**

* **Binary Search Trees (BSTs): Efficient searching, insertion, and deletion. BSTs are used in databases and file systems.**
* **Heaps: A type of binary tree used in implementing priority queues.**
* **Expression Trees: Used to represent arithmetic expressions where leaves are operands and internal nodes are operators.**
* **Huffman Coding Trees: Used in data compression algorithms to represent variable-length codes.**

**Create Tree**

**import java.util.\*;**

**public class createTree{**

**static Scanner sc;**

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

**create();**

**}**

**public static Node create(){**

**sc = new Scanner(System.in);**

**Node root = null;**

**System.out.println("Enter data : ");**

**int data = sc.nextInt();**

**root = new Node(data);**

**if(data == -1) return null;**

**System.out.println("Enter left data for : "+ data);**

**root.left = create();**

**System.out.println("Enter right data for : "+ data);**

**root.right = create();**

**return root;**

**}**

**}**

**class Node{**

**Node left, right;**

**int data;**

**public Node(int data){**

**this.data = data;**

**}**

**}**

Inorder, PreOrder, PostOrder

**public static void inOrder(Node root){**

**if(root==null){**

**return ;**

**}**

**inOrder(root.left);**

**System.out.print(root.data + " ");**

**inOrder(root.right);**

**}**

**public static void preOrder(Node root){**

**if(root == null){**

**return ;**

**}**

**System.out.print(root.data + " ");**

**preOrder(root.left);**

**preOrder(root.right);**

**}**

**public static void postOrder(Node root){**

**if(root == null){**

**return ;**

**}**

**postOrder(root.left);**

**postOrder(root.right);**

**System.out.print(root.data + " ");**

**}**

**Day 6**

**Array, String, LinkedList, Stack, Queue**

**Why Array indexing in Java starts from 0 ?**

**Array indexing in Java starts from 0 because it aligns with how memory addresses are calculated. The index represents an offset from the base address of the array. So, arr[0] directly refers to the first memory location without any additional offset, making it efficient. This approach is also consistent with the historical design of earlier programming languages like C.**

**1. Find the Second Largest Element in an Array**

* **Problem: Given an array of integers, find the second largest element in the array.**
* **Hint: Traverse the array once to find the largest element. Traverse it again to find the largest element that is smaller than the first largest element.**
* **Test Cases:**
  + **Input: [12, 35, 1, 10, 34, 1] Output: 34**
  + **Input: [10, 10, 10] Output: None (or a message indicating no second largest element exists)**
  + **Input: [5, 7, 2, 7, 3] Output: 5**

**2. Rotate Array**

* **Problem: Given an array, rotate the array to the right by k steps, where k is non-negative.**
* **Hint: Reverse the entire array, then reverse the first k elements, and finally reverse the remaining elements.**
* **Test Cases:**
  + **Input: nums = [1, 2, 3, 4, 5, 6, 7], k = 3 Output: [5, 6, 7, 1, 2, 3, 4]**
  + **Input: nums = [-1, -100, 3, 99], k = 2 Output: [3, 99, -1, -100]**
  + **Input: nums = [1, 2], k = 3 Output: [2, 1]**

**3. Find All Pairs with a Given Sum**

* **Problem: Given an array of integers, find all unique pairs in the array that sum up to a specific target number.**
* **Hint: Use a HashMap to store the difference between the target and the current element as you iterate through the array.**
* **Test Cases:**
  + **Input: nums = [2, 7, 11, 15, -2, 7], target = 9 Output: [(2, 7), (-2, 11)]**
  + **Input: nums = [1, 1, 2, 45, 46, 46], target = 47 Output: [(1, 46), (2, 45)]**
  + **Input: nums = [1, 2, 3, 4, 5], target = 10 Output: [] (no pairs found)**

**4. Merge Two Sorted Arrays**

* **Problem: Given two sorted arrays, merge them into a single sorted array.**
* **Hint: Use two pointers, one for each array, and compare elements to merge them into a new array.**
* **Test Cases:**
  + **Input: arr1 = [1, 3, 5, 7], arr2 = [2, 4, 6, 8] Output: [1, 2, 3, 4, 5, 6, 7, 8]**
  + **Input: arr1 = [5, 6, 7], arr2 = [1, 2, 3, 4] Output: [1, 2, 3, 4, 5, 6, 7]**
  + **Input: arr1 = [1, 2], arr2 = [] Output: [1, 2]**

**5. Find the Missing Number**

* **Problem: Given an array containing n distinct numbers in the range [0, n], find the one number that is missing from the array.**
* **Hint: The sum of the first n natural numbers is n\*(n+1)/2. Subtract the sum of the array from this value to find the missing number.**
* **Test Cases:**
  + **Input: [3, 0, 1] Output: 2**
  + **Input: [0, 1] Output: 2**
  + **Input: [9,6,4,2,3,5,7,0,1] Output: 8**

**6. Find the Maximum Subarray (Kadane’s Algorithm)**

* **Problem: Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.**
* **Hint: Use Kadane’s algorithm, which involves iterating through the array and at each position, determining the maximum subarray sum ending at that position.**
* **Test Cases:**
  + **Input: [-2,1,-3,4,-1,2,1,-5,4] Output: 6 (subarray [4,-1,2,1] has the maximum sum)**
  + **Input: [1] Output: 1**
  + **Input: [5,4,-1,7,8] Output: 23**

**7. Sort an Array of 0s, 1s, and 2s (Dutch National Flag Problem)**

* **Problem: Given an array with 0s, 1s, and 2s, sort it in linear time without using any extra space.**
* **Hint: Use three pointers to keep track of the current position, the next position for 0, and the next position for 2, and swap elements accordingly.**
* **Test Cases:**
  + **Input: [2, 0, 2, 1, 1, 0] Output: [0, 0, 1, 1, 2, 2]**
  + **Input: [2, 0, 1] Output: [0, 1, 2]**
  + **Input: [0, 0, 0, 1, 1, 2, 2, 2] Output: [0, 0, 0, 1, 1, 2, 2, 2]**

**8. Find the Intersection of Two Arrays**

* **Problem: Given two arrays, write a function to compute their intersection. The result should be an array of unique elements that appear in both arrays.**
* **Hint: Use a HashSet to store elements of one array, then iterate through the second array to check for common elements.**
* **Test Cases:**
  + **Input: nums1 = [4, 9, 5], nums2 = [9, 4, 9, 8, 4] Output: [9, 4]**
  + **Input: nums1 = [1, 2, 2, 1], nums2 = [2, 2] Output: [2]**
  + **Input: nums1 = [1, 2, 3, 4, 5], nums2 = [6, 7, 8, 9] Output: []**

**9. Find the Longest Consecutive Sequence**

* **Problem: Given an unsorted array of integers, find the length of the longest consecutive elements sequence.**
* **Hint: Use a HashSet to keep track of elements and check for each element if it is the start of a sequence.**
* **Test Cases:**
  + **Input: [100, 4, 200, 1, 3, 2] Output: 4 (the longest consecutive elements sequence is [1, 2, 3, 4])**
  + **Input: [0,3,7,2,5,8,4,6,0,1] Output: 9**
  + **Input: [9,1,4,7,3,-1,0,5,8,-1,6] Output: 7 (the longest consecutive elements sequence is [0, 1, 2, 3, 4, 5, 6])**

**10. Trapping Rain Water**

* **Problem: Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.**
* **Hint: Use two pointers to scan the elevation map from both ends and calculate trapped water by keeping track of the maximum heights encountered.**
* **Test Cases:**
  + **Input: [0,1,0,2,1,0,1,3,2,1,2,1] Output: 6**
  + **Input: [4,2,0,3,2,5] Output: 9**
  + **Input: [1,0,2] Output: 1**

**String**

**1. Reverse a String**

**Problem**: Given a string, reverse it.

**Example**:

* Input: "hello"
* Output: "olleh"

**2. Check if a String is a Palindrome**

**Problem**: Determine if a given string is a palindrome (reads the same forward and backward).

**Example**:

* Input: "racecar"
* Output: true
* Input: "hello"
* Output: false

**3. Longest Substring Without Repeating Characters**

**Problem**: Given a string, find the length of the longest substring without repeating characters.

**Example**:

* Input: "abcabcbb"
* Output: 3 (The longest substring is "abc")

**4. Find the First Non-Repeating Character**

**Problem**: Given a string, find the first non-repeating character in it.

**Example**:

* Input: "swiss"
* Output: 'w'

**5. Anagrams**

**Problem**: Check if two strings are anagrams of each other (i.e., they contain the same characters in different orders).

**Example**:

* Input: "listen", "silent"
* Output: true
* Input: "hello", "world"
* Output: false

**6. String Compression**

**Problem**: Implement basic string compression using the counts of repeated characters. If the compressed string is not smaller than the original string, return the original string.

**Example**:

* Input: "aabcccccaaa"
* Output: "a2b1c5a3"

**7. Longest Common Prefix**

**Problem**: Given an array of strings, find the longest common prefix string amongst them.

**Example**:

* Input: ["flower", "flow", "flight"]
* Output: "fl"

**8. Substring Search**

**Problem**: Find the first occurrence of a substring in a string.

**Example**:

* Input: ("hello", "ll")
* Output: 2 (The substring "ll" starts at index 2)

**9. String Permutations**

**Problem**: Given a string, generate all possible permutations of the characters in the string.

**Example**:

* Input: "abc"
* Output: ["abc", "acb", "bac", "bca", "cab", "cba"]

**10. Valid Parentheses**

**Problem**: Given a string containing just the characters '(', ')', '{', '}', '[', and ']', determine if the input string is valid (i.e., the brackets are balanced).

**Example**:

* Input: "()[]{}"
* Output: true
* Input: "(]"
* Output: false

**11. Replace Spaces**

**Problem**: Given a string, replace all spaces with %20.

**Example**:

* Input: "Mr John Smith ", with true for length 13
* Output: "Mr%20John%20Smith"

**12. Count and Say**

**Problem**: Generate the nth term of the count-and-say sequence.

**Example**:

* Input: 1
* Output: "1"
* Input: 4
* Output: "1211"

**13. Minimum Window Substring**

**Problem**: Given two strings s and t, find the minimum window in s which will contain all the characters in t (including duplicates).

**Example**:

* Input: "ADOBECODEBANC", "ABC"
* Output: "BANC"

**14. Check if Two Strings are Rotations**

**Problem**: Check if one string is a rotation of another.

**Example**:

* Input: "waterbottle", "erbottlewat"
* Output: true

**15. String to Integer (atoi)**

**Problem**: Implement the atoi function which converts a string to an integer.

**Example**:

* Input: "42"
* Output: 42
* Input: "-42"
* Output: -42

**Character class**

**Key Methods:**

1. **toUpperCase(char ch)**:
   * Converts the given character to uppercase.
   * Example: Character.toUpperCase('a') returns 'A'.
2. **toLowerCase(char ch)**:
   * Converts the given character to lowercase.
   * Example: Character.toLowerCase('A') returns 'a'.
3. **isDigit(char ch)**:
   * Determines if the given character is a digit.
   * Example: Character.isDigit('5') returns true.
4. **isLetter(char ch)**:
   * Determines if the given character is a letter.
   * Example: Character.isLetter('A') returns true.
5. **isWhitespace(char ch)**:
   * Determines if the given character is a whitespace character (e.g., space, tab).
   * Example: Character.isWhitespace(' ') returns true.
6. **isUpperCase(char ch)**:
   * Determines if the given character is an uppercase letter.
   * Example: Character.isUpperCase('A') returns true.
7. **isLowerCase(char ch)**:
   * Determines if the given character is a lowercase letter.
   * Example: Character.isLowerCase('a') returns true.
8. **isLetterOrDigit(char ch)**:
   * Determines if the given character is a letter or a digit.
   * Example: Character.isLetterOrDigit('A') returns true.
9. **isAlphabetic(int codePoint)**:
   * Determines if the given Unicode code point is alphabetic.
   * Example: Character.isAlphabetic('A') returns true.
10. **getNumericValue(char ch)**:
    * Returns the numeric value of the given character (if it's a digit or a letter that has a numeric value).
    * Example: Character.getNumericValue('3') returns 3.
11. **toTitleCase(char ch)**:
    * Converts the given character to title case using case mapping information from the Unicode data.
    * Example: Character.toTitleCase('a') could return 'A' in certain contexts.
12. **forDigit(int digit, int radix)**:
    * Returns the character representation of the given digit in the specified radix (base).
    * Example: Character.forDigit(10, 16) returns 'a'.
13. **charCount(int codePoint)**:
    * Returns the number of char values needed to represent the specified Unicode code point.
    * Example: Character.charCount(0x1F600) returns 2 because the code point is a surrogate pair.
14. **codePointAt(CharSequence seq, int index)**:
    * Returns the Unicode code point at the specified index of the CharSequence.
    * Example: Character.codePointAt("Hello", 1) returns 101 (Unicode for 'e').
15. **codePointBefore(CharSequence seq, int index)**:
    * Returns the Unicode code point before the specified index of the CharSequence.
    * Example: Character.codePointBefore("Hello", 1) returns 72 (Unicode for 'H').
16. **isSurrogate(char ch)**:
    * Determines if the given character is a surrogate character (part of a surrogate pair).
    * Example: Character.isSurrogate('\uD800') returns true.

**Day 7**

**Sliding Window**

**Day 12**

**Dynamic Programming**

**What is Dynamic Programming?**

Dynamic Programming (DP) is a powerful technique in computer science and programming used to solve complex problems by breaking them down into simpler subproblems. It is particularly useful when a problem can be divided into overlapping subproblems, and the solution to the problem can be constructed efficiently from the solutions to the subproblems.

**Top-Down Approach**

The **Top-Down Approach** in dynamic programming is also known as the **memoization** technique. In this approach, we start solving the problem from the top (the original problem) and break it down into smaller subproblems. We solve these subproblems and store their results in a table (usually an array) so that if the same subproblem is encountered again, we don't have to solve it again; instead, we can directly use the stored result.

**Memoization**

**Memoization** is a technique used to optimize the Top-Down Approach. It involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. This prevents the same calculation from being done multiple times, thus saving time.

**Example: Fibonacci Sequence**

The Fibonacci sequence is a classic example of where dynamic programming can be used. The sequence is defined as:

* F(0)=0
* F(1)=1
* F(n)=F(n−1)+F(n−2)F(n) for n≥2n

If we calculate Fibonacci numbers without using dynamic programming, we might end up recalculating the same Fibonacci numbers multiple times, which is inefficient.

**Code Explanation**

Here’s a Java implementation using the Top-Down Approach with memoization to compute the Fibonacci numbers:

class Solution {

static int fibonacciNumber(int n, int dp[]) {

// Base cases: if n is 0 or 1, return n

if (n <= 1) {

return n;

}

// If the result is already computed, return it

if (dp[n] != -1) {

return dp[n];

}

// Compute the nth Fibonacci number by solving subproblems

dp[n] = fibonacciNumber(n - 1, dp) + fibonacciNumber(n - 2, dp);

// Return the result

return dp[n];

}

static int nthFibonacci(int n) {

int dp[] = new int[n + 1]; // Array to store the results of subproblems

// Initialize all elements of dp array to -1 (indicating uncomputed results)

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

dp[i] = -1;

}

// Calculate the nth Fibonacci number

return fibonacciNumber(n, dp);

}

}

**How the Code Works:**

1. **Array Initialization**:
   * The dp array is initialized with -1 to indicate that no Fibonacci numbers have been computed yet.
2. **Base Case**:
   * If n is 0 or 1, the function simply returns n (because F(0) = 0 and F(1) = 1).
3. **Memoization Check**:
   * Before computing a Fibonacci number, the function checks if it has already been computed and stored in the dp array. If so, it returns the stored value, avoiding redundant computation.
4. **Recursive Computation**:
   * If the Fibonacci number hasn’t been computed yet, the function calculates it by recursively calling itself to solve the smaller subproblems (F(n-1) and F(n-2)). The result is then stored in the dp array.
5. **Result**:
   * The nthFibonacci function initializes the dp array and calls the fibonacciNumber function to get the nth Fibonacci number.

**Bottom-Up Approach in Dynamic Programming**

1. **Identify the Subproblem**:
   * Determine what the smallest subproblem is. Typically, this is the base case in the recursive approach.
   * Decide how the larger problem can be solved using smaller subproblems.
2. **Define the DP Table**:
   * Create a table (usually an array or matrix) to store the results of subproblems.
   * The size of this table is usually determined by the size of the input or the range of possible values for the problem.
3. **Initialize the Base Case(s)**:
   * Initialize the base cases in the DP table. These are the simplest subproblems that can be solved directly.
   * For example, in the Fibonacci problem, dp[0] = 0 and dp[1] = 1.
4. **Iteratively Fill the DP Table**:
   * Use a loop to fill in the DP table, starting from the base cases and building up to the original problem.
   * For each entry in the table, use the results of previous entries (which correspond to smaller subproblems) to compute the current entry.
5. **Return the Final Result**:
   * The final result for the original problem is usually found at the last entry of the DP table.
6. **(Optional) Optimize Space Complexity**:
   * If the problem only requires keeping track of a few previous results, consider reducing the space complexity by using a few variables instead of a full table.

**General Pseudocode Template**

function solveProblem(input):

# Step 1: Define the DP table

dp = new array of size [size determined by the problem]

# Step 2: Initialize base cases

dp[base\_case\_index] = base\_case\_value

# Step 3: Fill the DP table iteratively

for i from base\_case\_index + 1 to desired\_value:

dp[i] = combine(dp[smaller\_subproblem\_1], dp[smaller\_subproblem\_2], ..., dp[smaller\_subproblem\_k])

# Step 4: Return the final result

return dp[desired\_value]

**Example: Using the Template**

Let’s take the example of solving the problem of finding the minimum number of coins needed to make a certain amount:

// Function to find the minimum number of coins needed to make amount n

static int minCoins(int[] coins, int n) {

// Step 1: Define the DP table

int dp[] = new int[n + 1];

// Step 2: Initialize base cases

dp[0] = 0; // No coins are needed to make 0 amount

// Step 3: Fill the DP table iteratively

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

dp[i] = Integer.MAX\_VALUE;

for (int coin : coins) {

if (i - coin >= 0 && dp[i - coin] != Integer.MAX\_VALUE) {

dp[i] = Math.min(dp[i], dp[i - coin] + 1);

}

}

}

// Step 4: Return the final result

return dp[n] == Integer.MAX\_VALUE ? -1 : dp[n];

}

Practice Questions

1. **Climbing Stairs**  
You are climbing a staircase. It takes n steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

* Test Case 1:  
  Input: n = 2  
  Output: 2
* Test Case 2:  
  Input: n = 3  
  Output: 3

2. **House Robber**  
You are a professional robber planning to rob houses along a street. Each house has a certain amount of money. You cannot rob two adjacent houses. Find the maximum amount of money you can rob.

* Test Case 1:  
  Input: nums = [2,3,2]  
  Output: 3
* Test Case 2:  
  Input: nums = [1,2,3,1]  
  Output: 4

3. **Coin Change**  
You are given an array of coins and an amount. Find the minimum number of coins to make up the amount.

* Test Case 1:  
  Input: coins = [1, 2, 5], amount = 11  
  Output: 3
* Test Case 2:  
  Input: coins = [2], amount = 3  
  Output: -1

4. **Longest Increasing Subsequence**  
Given an integer array, return the length of the longest increasing subsequence.

* Test Case 1:  
  Input: nums = [10,9,2,5,3,7,101,18]  
  Output: 4
* Test Case 2:  
  Input: nums = [0,1,0,3,2,3]  
  Output: 4

5. **Knapsack Problem**  
Given weights and values of n items, determine the maximum value you can carry with a knapsack of capacity W.

* Test Case 1:  
  Input: weights = [1, 2, 3], values = [10, 20, 30], W = 50  
  Output: 60
* Test Case 2:  
  Input: weights = [1, 2, 3], values = [10, 20, 30], W = 3  
  Output: 30

6. **Edit Distance**  
Given two strings word1 and word2, return the minimum number of operations (insert, delete, replace) required to convert word1 to word2.

* Test Case 1:  
  Input: word1 = "horse", word2 = "ros"  
  Output: 3
* Test Case 2:  
  Input: word1 = "intention", word2 = "execution"  
  Output: 5

7. **Partition Equal Subset Sum**  
Given an array, check if it can be partitioned into two subsets such that the sum of the elements in both subsets is equal.

* Test Case 1:  
  Input: nums = [1, 5, 11, 5]  
  Output: true
* Test Case 2:  
  Input: nums = [1, 2, 3, 5]  
  Output: false

8. **Unique Paths**  
A robot is located at the top-left corner of an m x n grid. The robot can only move down or right. How many unique paths can the robot take to reach the bottom-right corner?

* Test Case 1:  
  Input: m = 3, n = 7  
  Output: 28
* Test Case 2:  
  Input: m = 3, n = 2  
  Output: 3

9. **Jump Game**  
Given an array of non-negative integers, determine if you are able to reach the last index starting from the first.

* Test Case 1:  
  Input: nums = [2,3,1,1,4]  
  Output: true
* Test Case 2:  
  Input: nums = [3,2,1,0,4]  
  Output: false

10. **Longest Palindromic Subsequence**  
Given a string s, find the longest subsequence which is a palindrome.

* Test Case 1:  
  Input: s = "bbbab"  
  Output: 4
* Test Case 2:  
  Input: s = "cbbd"  
  Output: 2

11. **Rod Cutting Problem**  
Given a rod of length n and prices for each piece length, find the maximum revenue you can get by cutting the rod and selling the pieces.

* Test Case 1:  
  Input: length = 8, prices = [1,5,8,9,10,17,17,20]  
  Output: 22
* Test Case 2:  
  Input: length = 4, prices = [1,5,8,9]  
  Output: 10

12. **Minimum Path Sum**  
Given a grid filled with non-negative numbers, find a path from top-left to bottom-right which minimizes the sum of all numbers along the path.

* Test Case 1:  
  Input: grid = [[1,3,1],[1,5,1],[4,2,1]]  
  Output: 7
* Test Case 2:  
  Input: grid = [[1,2,3],[4,5,6]]  
  Output: 12

13. **Wildcard Matching**  
Given an input string s and a wildcard pattern p, implement wildcard matching with support for '?' and '\*'.

* Test Case 1:  
  Input: s = "aa", p = "a"  
  Output: false
* Test Case 2:  
  Input: s = "aa", p = "\*"  
  Output: true

14. **Palindromic Partitioning**  
Given a string s, partition it such that every substring is a palindrome. Return the minimum cuts needed for a palindrome partitioning.

* Test Case 1:  
  Input: s = "aab"  
  Output: 1
* Test Case 2:  
  Input: s = "a"  
  Output: 0

15. **Integer Break**  
Given an integer n, break it into at least two positive integers such that the product of those integers is maximized. Return the maximum product.

* Test Case 1:  
  Input: n = 2  
  Output: 1
* Test Case 2:  
  Input: n = 10  
  Output: 36

**Day 13**

**1. What is a Linked List?**

* **Answer: A linked list is a linear data structure in which elements, called nodes, are not stored at contiguous memory locations. Each node contains two parts: the data and a reference (or pointer) to the next node in the sequence. Linked lists allow for efficient insertion and deletion of elements without the need to reallocate or reorganize the entire structure, unlike arrays.**

**2. What are the different types of Linked Lists?**

* **Answer:**
  + **Singly Linked List: Each node points to the next node and the last node points to null.**
  + **Doubly Linked List: Each node has two pointers, one pointing to the next node and the other to the previous node.**
  + **Circular Linked List: In this list, the last node points back to the first node, forming a circle.**
  + **Circular Doubly Linked List: Similar to a doubly linked list, but the last node points to the first node, and the first node points to the last node.**

**3. What are the advantages of a Linked List over an Array?**

* **Answer:**
  + **Dynamic Size: Linked lists can grow or shrink dynamically, while arrays have a fixed size.**
  + **Efficient Insertions/Deletions: Insertions and deletions can be done efficiently, especially at the beginning or middle of the list, as it only requires pointer adjustments. In arrays, these operations require shifting elements.**
  + **No Wastage of Memory: Linked lists do not require contiguous memory allocation, reducing memory wastage compared to arrays.**

**4. What are the disadvantages of Linked Lists?**

* **Answer:**
  + **Memory Usage: Each node in a linked list requires extra memory for the pointer/reference, which can lead to higher memory usage compared to arrays.**
  + **Traversal Time: Linked lists have a higher traversal time compared to arrays, as elements are not stored contiguously in memory, leading to cache misses.**
  + **No Random Access: Unlike arrays, linked lists do not support random access of elements, meaning you must traverse the list from the head to access an element.**

**5. How do you reverse a singly linked list?**

* **Answer: To reverse a singly linked list, you iterate through the list and change the next pointer of each node to point to its previous node. You maintain three pointers: previous, current, and next. Initially, previous is null, and current points to the head. As you traverse the list, you adjust the next pointer of the current node to point to previous, then move all three pointers one step forward. After the loop, set the head to previous.**

**6. What is the time complexity of inserting an element at the beginning of a linked list?**

* **Answer: The time complexity is O(1). Inserting at the beginning of a linked list only requires updating the head pointer to point to the new node, which is a constant-time operation.**

**7. How would you detect and remove a cycle in a linked list?**

* **Answer: To detect a cycle, you can use Floyd’s Cycle Detection algorithm (Tortoise and Hare). Two pointers, one moving twice as fast as the other, are used. If they meet, a cycle is present. To remove the cycle, after detecting it, reset one pointer to the head and move both pointers one step at a time. The node where they meet again is the start of the cycle, and the cycle can be removed by setting the next pointer of the previous node to null.**

**8. Explain the difference between a stack and a queue implemented using linked lists.**

* **Answer:**
  + **Stack: A stack is a LIFO (Last In, First Out) data structure where elements are added and removed from the same end, typically the top. When implemented using a linked list, the insertion and deletion operations (push and pop) are performed at the head.**
  + **Queue: A queue is a FIFO (First In, First Out) data structure where elements are added at one end (the rear) and removed from the other end (the front). When implemented using a linked list, insertions are done at the rear and deletions at the front.**

**9. What is the difference between a singly and doubly linked list?**

* **Answer:**
  + **Singly Linked List: Each node contains a single pointer pointing to the next node in the list.**
  + **Doubly Linked List: Each node contains two pointers, one pointing to the next node and one to the previous node, allowing traversal in both directions.**

**10. When would you prefer a linked list over an array?**

* **Answer: Linked lists are preferred when:**
  + **You need dynamic resizing of the data structure.**
  + **You need frequent insertions and deletions, especially at the beginning or middle of the list.**
  + **Memory is fragmented and cannot be allocated contiguously (arrays require contiguous memory).**

**11. What is a sentinel node in a linked list?**

* **Answer: A sentinel node is a dummy node that is used as a placeholder in a linked list to simplify the implementation of linked list operations. It does not hold any meaningful data. Sentinel nodes can be used as a head node in a singly linked list or as both head and tail in a doubly linked list to avoid edge cases.**

**12. How would you find the length of a linked list?**

* **Answer: To find the length of a linked list, you traverse the list from the head to the end, incrementing a counter at each step. The value of the counter at the end of the traversal will be the length of the list.**

**13. What is the time complexity of searching for an element in a linked list?**

* **Answer: The time complexity of searching for an element in an unsorted singly linked list is O(n), where n is the number of nodes in the list. This is because you may need to traverse the entire list to find the element.**

**14. What are some applications of linked lists?**

* **Answer:**
  + **Implementation of other data structures: Stacks, queues, and hash tables can be implemented using linked lists.**
  + **Dynamic memory allocation: Linked lists are used in dynamic memory allocation systems to manage free memory blocks.**
  + **Graph adjacency lists: Linked lists are used to represent adjacency lists in graph implementations.**
  + **Music/Video players: To implement the playlist feature where tracks are played sequentially.**

**15. What is the purpose of a dummy node in linked list problems?**

* **Answer: A dummy node is a placeholder node often used to simplify boundary conditions in linked list problems. It acts as a temporary starting point to avoid handling special cases for the head of the list separately. This is particularly useful in operations like merging, adding, or removing nodes from a list.**

**Questions That Ask in Companies**

**1. Reverse a Linked List**

* **Problem**: Reverse a singly linked list.
* **Variants**:
  + Reverse the entire linked list.
  + Reverse a linked list in groups of k nodes.

**2. Detect a Cycle in a Linked List**

* **Problem**: Determine if a cycle (loop) exists in a linked list.
* **Follow-up**: Find the starting node of the cycle if a cycle is present.
* **Algorithm**: Floyd’s Cycle Detection (Tortoise and Hare).

**3. Merge Two Sorted Linked Lists**

* **Problem**: Merge two sorted linked lists into a single sorted linked list.
* **Variants**:
  + Merge k sorted linked lists.

**4. Find the Middle of a Linked List**

* **Problem**: Find the middle node of a linked list.
* **Variant**: If there are two middle nodes, return the second one.

**5. Remove N-th Node from End of List**

* **Problem**: Remove the N-th node from the end of a linked list.
* **Variant**: Solve it in one pass.

**6. Intersection of Two Linked Lists**

* **Problem**: Find the node at which two singly linked lists intersect.
* **Follow-up**: What if the lists do not intersect?

**7. Remove Duplicates from a Sorted Linked List**

* **Problem**: Remove all duplicate nodes from a sorted linked list.
* **Variant**: Remove duplicates from an unsorted linked list (using a hash set).

**8. Palindrome Linked List**

* **Problem**: Check if a linked list is a palindrome.
* **Follow-up**: Solve it in O(n) time and O(1) space.

**9. Flatten a Multilevel Doubly Linked List**

* **Problem**: Given a doubly linked list where in addition to the next and previous pointers, each node has a child pointer, flatten the list.

**10. Rotate a Linked List**

* **Problem**: Given a linked list, rotate it to the right by k places.
* **Variant**: Rotate to the left by k places.

**11. Add Two Numbers Represented by Linked Lists**

* **Problem**: Two numbers are represented by two linked lists, where each node contains a single digit. The digits are stored in reverse order. Add the two numbers and return the sum as a linked list.
* **Variant**: Numbers are stored in non-reversed order.

**12. Clone a Linked List with Random Pointers**

* **Problem**: Each node in the linked list has two pointers: next and random. Write a function to clone the linked list with next and random pointers.

**13. Delete Node in a Linked List**

* **Problem**: Given a node in a singly linked list, delete that node without having access to the head of the list.
* **Note**: You’re only given the node to be deleted.

**14. Segregate Even and Odd Nodes**

* **Problem**: Rearrange the linked list so that all even nodes are placed before all odd nodes.

**15. Intersection of Two Sorted Linked Lists**

* **Problem**: Find the intersection of two sorted linked lists.

**16. Linked List Cycle II**

* **Problem**: Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

**17. Flatten a Linked List**

* **Problem**: Given a linked list where each node contains two pointers: right and down, flatten the linked list to a single level.
* **Note**: right represents the next node in the list, and down represents the next level of nodes.

**18. Reorder List**

* **Problem**: Reorder a given linked list to alternate between the first and last node, second and second last node, and so on.

**19. Partition List**

* **Problem**: Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

**20. Reverse Nodes in k-Group**

* **Problem**: Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.
* **Note**: k is a positive integer and is less than or equal to the length of the linked list.

**Day 14**

**1. Balanced Parentheses**

* **Problem: Check if a given string of parentheses is balanced.**
* **Example: {[()()]} is balanced, while {[(])} is not.**

**2. Implement a Stack using Arrays/Linked List**

* **Problem: Create a stack using an array or linked list. Implement the basic stack operations: push, pop, peek, and isEmpty.**

**3. Implement a Queue using Two Stacks**

* **Problem: Implement a queue using two stacks with enqueue and dequeue operations.**

**4. Next Greater Element**

* **Problem: Given an array, find the next greater element for each element in the array.**
* **Example: For the array [4, 5, 2, 10], the next greater elements are [5, 10, 10, -1].**

**5. Stock Span Problem**

* **Problem: Calculate the span of stock’s price for all days. The span of the stock’s price today is defined as the maximum number of consecutive days starting from today and going backwards, for which the price of the stock was less than or equal to today’s price.**
* **Example: For the prices [100, 80, 60, 70, 60, 75, 85], the spans are [1, 1, 1, 2, 1, 4, 6].**

**6. Min Stack**

* **Problem: Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.**
* **Example: Implement getMin() that returns the minimum element in the stack.**

**7. Largest Rectangle in Histogram**

* **Problem: Given an array of integers representing the histogram’s bar height where the width of each bar is 1, find the area of the largest rectangle in the histogram.**
* **Example: For the array [2, 1, 5, 6, 2, 3], the largest rectangle has an area of 10.**

**8. Evaluate Reverse Polish Notation**

* **Problem: Evaluate the value of an arithmetic expression in Reverse Polish Notation (RPN).**
* **Example: For the RPN expression ["2", "1", "+", "3", "\*"], the result is 9.**

**9. Sort a Stack**

* **Problem: Write a program to sort a stack such that the smallest items are on the top.**
* **Constraint: You can use only one additional stack.**

**10. Valid Parentheses**

* **Problem: Given a string containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.**
* **Example: (), ()[]{}[] are valid; ([)] is not.**

**11. Trapping Rain Water**

* **Problem: Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.**
* **Example: For the array [0,1,0,2,1,0,1,3,2,1,2,1], the trapped water is 6.**

**12. Remove K Digits**

* **Problem: Given a non-negative integer represented as a string, remove k digits from the number so that the new number is the smallest possible.**
* **Example: For the number 1432219 and k = 3, the smallest number is 1219.**

**13. Decode String**

* **Problem: Given an encoded string, return its decoded string. The encoding rule is: k[encoded\_string], where the encoded\_string inside the square brackets is being repeated exactly k times.**
* **Example: The string 3[a]2[bc] should return aaabcbc.**

**14. Flatten a Nested List Iterator**

* **Problem: Implement an iterator to flatten a nested list.**
* **Example: Given the list [[1,1],2,[1,1]], return [1,1,2,1,1].**

**15. Basic Calculator**

* **Problem: Implement a basic calculator to evaluate a simple expression string. The expression string contains only non-negative integers, +, -, \*, / operators and empty spaces.**
* **Example: For the expression "3+2\*2", the result is 7.**

1. **What is a stack?**
   * **A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. A stack allows operations only at one end, called the "top" of the stack. Operations supported by a stack typically include push (to add an element to the top), pop (to remove the element from the top), peek or top (to view the element at the top without removing it), and isEmpty (to check if the stack is empty).**
2. **What are the main operations of a stack?**
   * **push(data): Adds an element to the top of the stack.**
   * **pop(): Removes the element from the top of the stack and returns it. If the stack is empty, this operation may return an error or indicate that the stack is empty.**
   * **peek() or top(): Returns the element at the top of the stack without removing it. If the stack is empty, this operation may return an error or indicate that the stack is empty.**
   * **isEmpty(): Checks if the stack is empty and returns true if it is, otherwise returns false.**
3. **What is the LIFO principle in a stack?**
   * **LIFO stands for Last-In-First-Out. This principle means that the most recently added element (the last one pushed onto the stack) is the first one to be removed (the first one popped off the stack). This is akin to a stack of plates where you add and remove plates from the top of the stack.**
4. **What are the use cases of a stack?**
   * **Function Call Management: Stacks are used to manage function calls in programming languages. Each function call is pushed onto the call stack, and when the function completes, it is popped off the stack.**
   * **Expression Evaluation: Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation) and to convert infix expressions to postfix notation.**
   * **Undo Mechanisms: Many applications use stacks to implement undo features. Each action is pushed onto a stack, and an undo operation pops the last action off the stack.**
   * **Backtracking Algorithms: Stacks are used in algorithms that need to backtrack, such as depth-first search (DFS) in graph traversal or solving puzzles like mazes.**
   * **Parsing Syntax: Stacks are used in parsing expressions and validating parentheses in programming languages.**
5. **What is the difference between a stack and a queue?**
   * **A stack follows the Last-In-First-Out (LIFO) principle, meaning that the last element added is the first one removed. In contrast, a queue follows the First-In-First-Out (FIFO) principle, where the first element added is the first one removed. Stacks use operations like push and pop, while queues use operations like enqueue (to add) and dequeue (to remove).**
6. **How is a stack typically implemented?**
   * **A stack can be implemented using an array or a linked list. With an array, you maintain an index to keep track of the top of the stack. With a linked list, you use a node-based structure where each node points to the next node, and the top of the stack is represented by the head node.**
7. **What is a stack overflow and underflow?**
   * **Stack Overflow occurs when a stack exceeds its allocated size or capacity, often due to too many push operations. Stack Underflow happens when an attempt is made to pop or peek from an empty stack, indicating that no elements are available to remove or view.**
8. **Explain how to implement a stack using two queues.**
   * **To implement a stack using two queues, you can use the following approach:**
     + **Push Operation: Enqueue the new element into the first queue. Then, move all elements from the first queue to the second queue, so the new element becomes the front of the second queue. Swap the names of the two queues.**
     + **Pop Operation: Dequeue the front element from the first queue (which is now the stack's top).**
9. **What is the time complexity of stack operations?**
   * **The time complexity of standard stack operations (push, pop, and peek) is O(1), meaning they operate in constant time. This efficiency is because these operations involve only modifying or accessing the top of the stack.**
10. **How can a stack be used to reverse a string?**
    * **To reverse a string using a stack, push each character of the string onto the stack. Then, pop characters from the stack and concatenate them to get the reversed string. This works because the stack's LIFO nature reverses the order of the characters.**
11. **How can you check for balanced parentheses using a stack?**
    * **To check for balanced parentheses, iterate through each character in the string. Push each opening parenthesis onto the stack. For each closing parenthesis, check if the stack is not empty and if the top of the stack is the matching opening parenthesis. If it matches, pop the top of the stack. At the end of the iteration, if the stack is empty, the parentheses are balanced; otherwise, they are not.**
12. **Can you use a stack to evaluate arithmetic expressions?**
    * **Yes, a stack can be used to evaluate arithmetic expressions, particularly in postfix (Reverse Polish Notation) notation. Operands are pushed onto the stack. When an operator is encountered, operands are popped from the stack, the operation is performed, and the result is pushed back onto the stack. The final result will be on top of the stack.**