**JAVA ANSWERS**

1. **What is java?**

* Java is a high-level, class-based, object-oriented programming language.
* It was originally developed by Sun Microsystems and released in 1995

Java is widely used for building:

* Web applications
* Mobile apps (especially Android apps)
* Enterprise software
* Desktop applications
* Embedded systems

1. **What are the features of Java?**

* High level
* Robust
* Object-oriented
* Secure
* Dynamic
* Platform independent
* Portable
* High performance
* Multithreaded
* Architectural neutral
* Interpreted
* Simple
* Distributed

1. **What is java environment?**

The **Java Environment** refers to the set of tools, libraries, and runtime components required to develop and run Java applications. It consists of three main parts:

* **JDK ( Java development kit)**- full-featured software development kit used to write, compile, and run Java programs.
* **JRE ( Java runtime environment)-** A package of software needed to run Java programs.
* **JVM (Java virtual machine)**- The part of the Java environment that actually executes the Java bytecode.

1. **Exp the types of java applications**

* **Desktop Applications (Stand-alone)-** Software that runs on personal computers or laptops.
* **Web Applications-** Applications that run on servers and are accessed via web browsers.
* **Enterprise Applications-** Large-scale applications designed for business needs (high performance, scalability, security).
* **Mobile Applications-** Apps that run on mobile devices.
* **Embedded Systems-** Software that runs on devices with dedicated functions.
* **Cloud-Based Applications-** Apps that run and scale on cloud platforms.

1. **What is a class file in Java**

In Java, a class is a blueprint or template for creating objects. It defines:

* Properties (fields/variables)
* Behaviors (methods/functions)

1. **What happens at compile time?**

* At compile time, the code becomes .class.
* This .class file contains bytecode, which is executed by the JVM.

1. **What is variable in java and exp with eg?**

In **Java**, a **variable** is a container that holds data that can be changed during the execution of a program. Each variable has a **data type**, which determines the kind of data it can store, such as integers, decimals, characters, etc.

Syntax: dataType variableName = value;

Eg: int x=3;

* dataType: Type of the data (e.g., int, double, String)
* variableName: Name of the variable
* value: (Optional) Initial value assigned

1. **Explain the types of variable**

**Types of Variables in Java:**

* **Local Variable** – A **local variable** is declared inside a method, constructor, or block and is only accessible within it.
* **Instance Variable** – An **instance variable** is declared inside a class but outside any method. It belongs to the **object** of the class.
* **Static Variable** – A **static variable** is shared among all objects of a class. It is declared using the static keyword.

**Local variable eg**:

public class LocalExample {

public void display() {

int number = 10; // Local variable

System.out.println("Local value: " + number);

}

}

**Instance variable eg:**

public class InstanceExample {

int age = 25; // Instance variable

public void showAge() {

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

}

}

**Static variable eg:**

public class StaticExample {

static String college = "ABC College"; // Static variable

String name;

StaticExample(String n) {

name = n;

}

public void show() {

System.out.println(name + " studies at " + college);

}

}

1. **Exp the data types**

Java has 2 main types of data. They are:

* Primitive data type
* Non primitive data type

**PRIMITIVE DATA TYPE**

|  |  |  |  |
| --- | --- | --- | --- |
| **S NO** | **DATA TYPE** | **DESCRIPTION** | **EXAMPLE** |
| **1** | **Byte** | **1 byte** | **byte b = 100;** |
| **2** | **Int** | **2 byte** | **int age = 25;** |
| **3** | **Short** | **2 bytes** | **short s = 10000;** |
| **4** | **Long** | **8 byte** | **Long l= 123456L;** |
| **5** | **Boolean** | **True or false** | **boolean isOn = true;** |
| **6** | **Float** | **4 byte** | **float f = 5.75f;** |
| **7** | **char** | **2 byte** | **char c = 'A';** |
| **8** | **double** | **8 byte** | **double d = 19.99;** |

**NON PRIMITIVE DATA TYPE**

These include:

* String → String name = "Java";
* Arrays → int[] arr = {1,2,3};
* Classes, Interfaces, etc.

1. **Exp the operators in java? With eg**

The types of operators in java are:

* **Arithmetic operators**- Used to perform basic math operations like +, -, \*, /, %

**Eg: int result = 5 + 3; // result = 8**

* **Logical operators-** Used to combine boolean expressions.( &&, ||, ! )

**Eg: if (a > 5 && b < 10) { }**

* **Relational operators**- Compare two values and return a boolean.( ==, !=, >, <, >=, <=)

**Eg: if (a == b) { }**

* **Bitwise operators**- Operate on individual bits of integer values.( &, |, ^, ~, <<, >>)

**Eg: int x = 5 & 3; // x = 1**

* **Ternary operators**- Shorthand for if-else decision making.(condition ? value\_if\_true : value\_if\_false)

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

* **Assignment operators**- Assign values and support shorthand forms.( =, +=, -=, \*=, /=, %=)

**Eg: x += 5; // x = x + 5**

* **Shift operators-** Shift bits to the left or right.( <<, >>, >>>)

**Eg: int x = 8 << 1; // x = 16**

* **Unary operators**- Work with a single operand.( +, -, ++, --, !)

**Eg: a++; // increment a by 1**

1. **What are java key words**

**Keywords** are reserved words in Java with special meaning. You **cannot use them as variable names**.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| else | abstract | char | goto | long | protected | super | throws | catch |
| enum | assert | case | continue | try | public | switch | const | double |
| byte | boolean | int | short | for | requires | return | void | interface |
| final | extends | var | import | new | synchronised | package | volatile | private |
| finally | byte | if | static | null | implements | throw | while | strictfp |
| float | class | do | default | this | instanceof | module | native | transient |

1. **What is decision making in java**

**Decision making** allows a program to choose between different actions based on conditions.

**Common decision-making statements:**

* If - Executes a block **only if** the condition is true.
* if-else - Chooses between **two blocks** of code.
* if-else-if - Checks **multiple conditions** in order.
* switch- Used for **multiple exact value** checks.
* ternary (?) operator- Short form of if-else in one line.

1. **What is switch statement?**

A **switch statement** is a control structure used to execute one block of code among many **based on the value of a variable or expression**. It's often cleaner than writing multiple if-else statements.

Example:

**int day = 2;**

**switch (day) {**

**case 1:**

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

**break;**

**case 2:**

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

**break;**

**default:**

**System.out.println("Other day");**

**}**

1. **What is if statement? Explain all types of if statement with syntax**

The if statement is a control structure that allows a program to execute certain code only when a condition is true. It is the most basic decision-making statement in Java.

**Types of if Statements in Java:**

* Simple if statement
* If-else statement
* If else if ladder
* Nested if statement

**Simple if statement**: The if statement is a control structure that allows a program to execute certain code only when a condition is true. It is the most basic decision-making statement in Java.

**if statement syntax**

if (condition) {

// code to execute if condition is true

}

**Eg:**

if (a > b) {

System.out.println("A is greater");

}

**If-else statement**: The if-else statement executes one block of code if the condition is true, and another block if the condition is false.

**If else statement syntax:**

if (condition) {

// true block

} else {

// false block

Eg:

int number = 15;

if (number % 2 == 0) {

System.out.println("The number is even.");

} else {

System.out.println("The number is odd.");

}

**If-else-if statement**: The if-else-if ladder is used when you want to test multiple conditions one by one. The first true condition's block is executed.

**If else if statement syntax:**

if (condition1) {

// block 1

} else if (condition2) {

// block 2

} else {

// default block

}

**eg:**

int marks = 75;

if (marks >= 90) {

System.out.println("Grade A");

} else if (marks >= 75) {

System.out.println("Grade B");

} else if (marks >= 60) {

System.out.println("Grade C");

} else {

System.out.println("Fail");

}

**Nested if statement:** A nested if is an if statement inside another if. It is used when you need to check one condition inside another

**Syntax:**

if (condition1) {

if (condition2) {

// code if both condition1 and condition2 are true

}

}

**Eg:**

int age = 25;

int weight = 70;

if (age >= 18) {

if (weight >= 60) {

System.out.println("Eligible for army recruitment");

}}

1. **Diff between switch statement and if else if statement**

|  |  |  |
| --- | --- | --- |
| **S NO** | **switch** | **If else if** |
| **1** | **Used for exact value matching** | **Used for range-based or complex conditions** |
| **2** | **Can evaluate only specific values like int, char, String, enum** | **Can evaluate boolean expressions, ranges, and logical operators** |
| **3** | **Cleaner and more readable for multiple exact values** | **Less readable when there are many conditions** |
| **4** | **Can fall through if the break is not used** | **No fall-through; only one block runs** |
| **5** | **Handled with default** | **Handled with else** |
| **6** | **Example:**  **int marks = 85;**  **if (marks >= 90)**  **{**  **System.out.println("Grade A");**  **} else if (marks >= 75) {**  **System.out.println("Grade B");**  **} else {**  **System.out.println("Grade C or below");**  **}** | **example:**  **int day = 2;**  **switch (day) {**  **case 1:**  **System.out.println("Monday");**  **break;**  **case 2:**  **System.out.println("Tuesday");**  **break;**  **default:**  **System.out.println("Other day");**  **}** |

1. **What is for loop?**

A for loop is a control structure used to repeat a block of code a specific number of times. It consists of three parts: initialization, condition, and increment/decrement.

Eg:

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

System.out.println(i);

}

1. **What is for each loop?**

A for-each loop (also called enhanced for loop) is used to iterate over elements in arrays or collections without using an index.

Eg:

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

for (int num : numbers) {

System.out.println(num);

}

1. **What is while loop?**

A while loop executes a block of code as long as a specified condition is true.

Eg:

int i = 0;

while (i < 5) {

System.out.println(i);

i++;

}

1. **What is do-while loop?**

A do-while loop is similar to a while loop, but it runs the code block at least once, even if the condition is false.

Eg:

int i = 0;

do {

System.out.println(i);

i++;

} while (i < 5);

1. **What is java break statement?**

The break statement is used to exit a loop or switch statement early, before it naturally finishes.

Eg:

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

if (i == 3) break;

System.out.println(i);

}

1. **What is java continue statement?**

The continue statement skips the current iteration and goes to the next iteration of the loop.

Example:

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

if (i == 3) continue;

System.out.println(i);

}

1. **Difference between break and continue statements.**

|  |  |  |
| --- | --- | --- |
| **S NO** | **BREAK** | **CONTINUE** |
| **1** | **Exits the loop entirely** | **Skips to the next loop iteration** |
| **2** | **Used to stop the loop early** | **Used to skip a certain case in loop** |
| **3** | **Exit on error or condition match** | **Skip unwanted values** |
| **4** | **Control jumps to the statement immediately following the loop or switch.** | **Control jumps to the next iteration’s condition check or increment step.** |
| **5** | **Applicable for All loops (for, while, do-while) and switch statements.** | **Only applicable inside loops (for, while, do-while), not switch.** |

1. **Why do we use comments in code?**

Comments are used to:

* Explain what the code is doing
* Improve code readability
* Help others (or yourself later) understand the logic
* Temporarily disable code during testing

1. **Explain the types of comments in java?**

Java supports three types of comments:

* Single-line comment: this uses //

**// This is a single-line comment**

* Multiline comment : this uses /\* \*/

**/\* This is a**

**multi-line comment \*/**

* Documentation comment: This uses /\*\* \*/, typically for generating JavaDocs.

**/\*\***

**\* This method adds two numbers**

**\* @param a First number**

**\* @param b Second number**

**\* @return Sum of a and b**

**\*/**

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

**return a + b;**

**}**

1. **What is oops? Why are we using it?**

OOPs (Object-Oriented Programming) is a programming paradigm that uses objects and classes to structure software.

We use oops for the following reasons:

* **Modularity** – Code is organized into reusable objects.
* **Reusability** – Inheritance reduces redundant code.
* **Maintainability** – Easier to update and debug structured code.
* **Scalability** – Objects can be extended without rewriting entire code.
* **Real-world Modeling** – Represents real-life entities effectively.

1. **What are the core components of oops?**

* Class – A blueprint for creating objects (e.g., Car class).
* Object – An instance of a class (e.g., BMW, Toyota).
* Encapsulation – Hiding internal data and exposing only necessary functions.
* Inheritance – Reusing code by creating parent-child class relationships.
* Polymorphism – One function/method behaving differently in different contexts.
* Abstraction – Simplifying complex systems by hiding unnecessary details.

1. **What is class in java?**

* A class in Java is a blueprint or template used to create objects. It defines the properties (attributes) and behaviors (methods) that objects of that class will have.
* Defined using the class keyword.
* Follows PascalCase naming convention (e.g., Car, BankAccount).

1. **What are the class declarations?**
   * **Access Modifiers**- Control where the class can be accessed from:

* public → Accessible from anywhere.
* default (no modifier) → Accessible only within the same package.
* private & protected → Not used for top-level classes (only for nested classes).
  + **Non-Access Modifiers**- Define additional behavior
* final → Cannot be subclassed (e.g., public final class Math).
* abstract → Cannot be instantiated (must be extended).
* static → Only for nested classes (not top-level).
  + **Body of the Class** – it is enclosed by ({})

Contains:

* Fields (Variables) → Store data (String name;).
* Constructors → Initialize objects (public Car() {…}).
* Methods → Define behavior (void startEngine()).
* Blocks → Static (static {…}) or instance ({…}) initialization.
* Nested Classes/Interfaces → Inner classes.

1. **What is object in java?**

An object in Java is an instance of a class that represents a real-world entity. It contains:

* State (data) → Stored in variables (fields)
* Behavior (actions) → Defined by methods

1. **What are the core components of object?**

An object in Java has three main components:

* State (Fields/Attributes) → Represented by variables (data).
* Behavior (Methods) → Represented by functions (actions).
* Identity (Unique Reference) → The memory address where the object is stored.

1. **What is data abstraction?**

Data Abstraction is an OOPs concept that hides complex implementation details and exposes only essential features to the user. It focuses on what an object does rather than how it does it.

* Achieved via:
  + **Abstract Classes** (partial abstraction, extends).
  + **Interfaces** (full abstraction, implements).

1. **What is encapsulation?**

* Encapsulation is an OOPs concept that binds data (variables) and methods (functions) into a single unit (class) and restricts direct access to internal data.
* Declare fields as private → Restricts direct access.
* Provide public getter/setter methods → Controlled access/modification.

1. **What is inheritance?**

Inheritance is a fundamental OOP concept in Java where a child class (subclass) inherits properties and behaviors from a parent class (superclass). It promotes code reusability and establishes an "is-a" relationship between classes.

Eg:

class ParentClass {

// Fields and methods

}

class ChildClass extends ParentClass {

// Inherits ParentClass members + can add new ones

}

1. **What is polymorphism?**

Polymorphism is a core OOP concept that allows one action to be performed in different ways. It means "many forms" and enables flexibility in Java programs.

1. **Diff between method overloading and method overriding**

|  |  |  |
| --- | --- | --- |
| **S NO** | **METHOD OVERLOADING** | **METHOD OVERRIDING** |
| **1** | **Same method name, different parameters.** | **Same method name and signature in Parent and child class** |
| **2** | **Compile-time (static binding)** | **Runtime (dynamic binding)** |
| **3** | **Increases method flexibility (multiple behaviors).** | **Modifies/extends parent class behavior.** |
| **4** | **Can have any access level.** | **Cannot be more restrictive than parent’s method.** |
| **5** | **It does not require inheritance** | **It requires inheritance** |

1. **What are the disadv of oops?**

* Steeper Learning Curve – Complex concepts like inheritance and polymorphism can be challenging for beginners.
* Performance Overhead – Objects consume more memory and CPU than procedural code.
* Over-Engineering – Can make simple programs unnecessarily complex.
* Slower Execution – Dynamic method dispatch (polymorphism) adds runtime overhead.
* Not Suitable for All Problems – Some tasks (e.g., mathematical computations) are better handled procedurally.
* Memory Consumption – More objects = higher RAM usage.

1. **What are the adv of oops?**

* Modularity – Breaks code into reusable, organized objects.
* Reusability – Inheritance and composition reduce redundant code.
* Data Security – Encapsulation protects data with private fields and public methods.
* Flexibility – Polymorphism allows multiple behaviors under one interface.
* Easy Maintenance – Isolated changes in classes minimize system-wide impact.
* Scalability – New features can be added without rewriting existing code.

1. **What are the core features of java 8?**

Java 8 introduced several major features:

* Lambda Expressions – Enables functional programming by allowing anonymous functions.
* Functional Interfaces – Interfaces with a single abstract method (SAM).
* Stream API – Supports functional-style operations on collections.
* Default Methods – Allows adding new methods to interfaces without breaking existing implementations.
* Method References – Shorthand syntax for calling methods using ::.
* Optional Class – Helps avoid NullPointerException by wrapping nullable values.
* New Date & Time API (java.time) – Improved APIs for date/time handling (LocalDate, LocalTime, etc.).

1. **What is a lambda expression and explain its key feature**

A lambda expression is an anonymous function that allows concise implementation of functional interfaces.

Key Features:

* Concise syntax – Reduces boilerplate code.
* Functional programming support – Enables passing behavior as a method argument.
* Type inference – Compiler infers parameter types automatically.

1. **What are the 3 lambda expressions and exp with eg**

* No Parameter Lambda- no parameters are specified

Eg: () -> System.out.println("Hello");

* Single Parameter Lambda- only 1 parameter is specified

Eg: (x) -> x \* x;

* Multiple Parameters Lambda- more than 1 parameter is specified

Eg: (a, b) -> a + b;

1. **What is java functional interfaces ?**

A functional interface is an interface with only one abstract method (SAM). Lambda expressions can be used to implement them.

Eg:

@FunctionalInterface

interface Greeting {

void sayHello(String name); // Single abstract method

}

Greeting greet = name -> System.out.println("Hello, " + name);

greet.sayHello("Alice"); // Output: Hello, Alice

1. **Explain all the functional interfaces with eg**

Java 8 provides built-in functional interfaces in java.util.function:

**1. Predicate<T>**

* Purpose: Tests a condition and returns boolean.
* Eg:

Predicate<Integer> isEven = num -> num % 2 == 0;

System.out.println(isEven.test(4)); // true

System.out.println(isEven.test(5)); // false

**2. Function<T, R>**

* Purpose: Takes an input (T) and returns a result (R).

Eg:

Function<String, Integer> strLength = str -> str.length();

System.out.println(strLength.apply("Hello")); // 5

**3. Consumer<T>**

* Purpose: Performs an operation without returning a value.
* Example:

Consumer<String> printUpperCase = s -> System.out.println(s.toUpperCase());

printUpperCase.accept("hello"); // HELLO

**4. Supplier<T>**

* Purpose: Supplies a value (no input, returns T).
* Example:

Supplier<Double> randomValue = () -> Math.random();

System.out.println(randomValue.get()); // e.g., 0.7423

**5. UnaryOperator<T>**

* Purpose: A Function where input and output types are the same (T → T).
* Eg:

UnaryOperator<Integer> square = x -> x \* x;

System.out.println(square.apply(5)); // 25

1. **What is java stream**

A Stream is a sequence of elements supporting sequential and parallel operations on collections.

Key Features:

* Not a data structure – Does not store data.
* Lazy evaluation – Operations are performed only when needed.
* Functional-style operations – Supports map, filter, reduce, etc.

Stream Operations:

* Intermediate ops: filter(), map(), sorted()
* Terminal ops: collect(), forEach(), reduce()

1. **What is exception?**

An exception is an unexpected event that occurs during program execution, disrupting the normal flow of instructions.

Examples of Exceptions:

* Dividing by zero → ArithmeticException
* Accessing an invalid array index → ArrayIndexOutOfBoundsException
* Reading a non-existent file → FileNotFoundException
* Null object reference → NullPointerException

1. **What is exception handling in java?**

Exception handling is a mechanism to manage runtime errors so that the program can:

* Continue executing instead of crashing.
* Provide meaningful error messages to users.
* Log errors for debugging

1. **What are the keywords used in exception handling**

Java provides five keywords to handle exceptions gracefully:

**Try**- Defines a block of code where exceptions might occur.

**Catch**- Catches and handles exceptions thrown in the try block.

**Finally**- Executes always (for cleanup like closing files).

**Throw**- Manually throws an exception.

**Throws**- Declares exceptions a method might throw (used in method signature).

1. **Exp java exception hierarchy**

The Java exception hierarchy is rooted at the Throwable class, with two main branches:

**JAVA EXCEPTION HIERARCHY**

**ERRORS  
(for unrecoverable JVM/system failures)  
EG:**

**OutOfMemoryError, StackOverflowError**

**EXCEPTIONS**

**CHECKED EXCEPTIONS  
[COMPILE TIME]  
EG:**

**IOException, SQLException**

**UNCHECKED EXCEPTIONS  
[RUN TIME]  
EG:  
NullPointerException, ArithmeticException**

1. **What are checked exception in java**

Checked exceptions are verified at compile-time. They must be:

* Caught in a try-catch block or
* Declared in the method signature using throws

Examples:

* FileNotFoundException (file access issues)
* ClassNotFoundException (class not found during reflection)
* InterruptedException (thread interruption)

1. **What are unchecked exception in java**

Unchecked exceptions (subclasses of RuntimeException) occur due to programming errors and are not checked at compile-time.

Examples:

* NullPointerException (accessing null references)
* ArrayIndexOutOfBoundsException (invalid array index)
* IllegalArgumentException (invalid method arguments)

1. **Difference between checked and unchecked exception**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Checked Exceptions** | **Unchecked Exceptions** |
| **1. Compile-Time Check** | **Enforced by compiler (must be handled/declared)** | **Not enforced (runtime-only)** |
| **2.Inheritance** | **Subclass of Exception (but not RuntimeException)** | **Subclass of RuntimeException** |
| **3. Recovery** | **Expected scenarios (e.g., file not found, network errors)** | **Programming errors (e.g., null pointer, invalid array access)** |
| **4. Handling Requirement** | **Mandatory (compile error if ignored)** | **Optional (but recommended for robustness)** |
| **5. Examples** | **IOException, SQLException,**  **ClassNotFoundException** | **NullPointerException, ArithmeticException, ArrayIndexOutOfBoundsException** |

1. **Difference between error and exception**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Errors (Error)** | **Exceptions (Exception)** |
| **1. Type** | **Represents unrecoverable system-level problems (JVM failures)** | **Represents recoverable application-level problems** |
| **2.Handling** | **Not meant to be caught/handled (though technically possible)** | **Must be handled (checked) or optional (unchecked)** |
| **3. Recovery** | **Cannot be recovered (program termination likely)** | **Can be recovered (graceful fallback possible)** |
| **4. Origin** | **Thrown by JVM (e.g., out of memory, stack overflow)** | **Thrown by application code or Java API** |
| **5. Examples** | **OutOfMemoryError, StackOverflowError, VirtualMachineError** | **Checked: IOException Unchecked: NullPointerException** |

1. **What is finally block?**

The finally block always executes (even if an exception occurs or a return statement is reached). It’s used for cleanup (e.g., closing files, releasing resources).

1. **What is the purpose of throw and throws exception**

**throw**: Manually trigger an exception (e.g., validate input):

if (age < 18) throw new IllegalArgumentException("Age must be 18+");

**throws**: Declares exceptions a method might throw (forces callers to handle them):

void readFile() throws IOException { ... }

1. **How can we catch multiple exceptions?**

There are 3 ways to catch multiple exceptions in Java:

**1. Multiple catch Blocks**

Handle each exception type separately.

**Eg:**

try {

// Risky code

} catch (IOException e) {

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

} catch (SQLException e) {

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

} catch (Exception e) {

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

}

**2. Multi-Catch Block**

Handle multiple exceptions in a single catch block using |.

**Eg:**

try {

// Risky code

} catch (IOException | SQLException e) { // Combined catch

System.out.println("Database/File Error: " + e.getMessage());

} catch (Exception e) {

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

}

**3. Catch Parent Exception**

Handle all exceptions with a superclass (e.g., Exception).

**Eg:**

try {

// Risky code

} catch (Exception e) { // Catches ALL exceptions

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

}

1. **What are the adv of exception**

* Prevents Program Crashes - Gracefully handles errors without abrupt termination.
* Improves Code Readability - Separates error-handling logic from business logic.
* Enables Debugging - Stack traces help identify root causes of failures.
* Supports Recovery - Allows retry mechanisms or alternative workflows.
* Maintains Program Flow - Continues execution after handling exceptions.

1. **What is a custom exception?**

* Custom exceptions are user-defined exception classes that extend Java's built-in exception hierarchy. They are used to represent application-specific error conditions.
* We use custom exception for better debugging , domain specific errors, forced handling and consistency

1. **Expl the below terminologies:**

Try block-  Code where exceptions might occur.

Catch block- Handles specific exceptions.

Finally block- Cleanup code (always executes).

Custom exception- User-defined exception class.

1. **Why an exception occur?**

* Logical errors: Divide by zero (ArithmeticException).
* Invalid input: Parsing a non-number (NumberFormatException).
* Resource issues: Missing file (FileNotFoundException).
* External failures: Database disconnect (SQLException).

1. **Exp the below terminologies:**

* ArithmeticException: Math errors (e.g., 10 / 0).
* ClassCastException: Invalid casting (e.g., String s = (String) new Object()).
* NullPointerException: Calling methods on null.
* ArrayIndexOutOfBoundsException: Accessing arr[10] in a 5-element array.
* ArrayStoreException: Storing a String in an Integer[].
* IllegalThreadStateException: Starting a thread twice.

1. **What is a user user-defined exception**

A user-defined exception is a custom exception class created by the programmer to handle application-specific errors. These exceptions extend Java's built-in Exception or RuntimeException classes.

Examples:

* InvalidPasswordException → "Password must contain a number!" (Checked)
* DatabaseConnectionException → "Failed to connect to DB!" (Checked)
* NegativeSalaryException → "Salary cannot be negative!" (Unchecked)

1. **What is file handling in java?**

* File handling in Java refers to reading from and writing to files stored on a system.
* Java provides classes in the java.io and java.nio packages to perform file operations like create, read, write, and delete files.

1. **Why is file handling is required?**

* File handling is required to store data permanently (outside the program).
* It allows saving user data, reading configuration files, generating logs, and exchanging information between programs.

1. **What is streams in java?**

* A stream in Java is a sequence of data. Java uses streams to perform input and output (I/O) operations. There are two types:
* InputStream for reading data
* OutputStream for writing data.

1. **Difference between text file and binary file?**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Text File** | **Binary File** |
| **1. Data Format** | **Stores data in human-readable characters** | **Stores data in machine-readable binary form** |
| **2. File Extension** | **Usually .txt, .csv** | **Usually .bin, .dat, .exe** |
| **3. Readability** | **Can be opened and read with a text editor** | **Cannot be read easily without a program** |
| **4. Data Storage** | **Stores data as plain text (ASCII/Unicode)** | **Stores data as raw bytes** |
| **5. File Size** | **Generally larger for same content** | **More compact and efficient** |
| **6. Use Case** | **Ideal for logs, configs, simple data storage** | **Used for images, videos, or compiled code** |
| **7. Processing Speed** | **Slower to process due to encoding/decoding** | **Faster for reading and writing operations** |

1. **How do you open and read the content of a text file in java?**

import java.io.BufferedReader;

import java.io.FileReader;

BufferedReader reader = new BufferedReader(new FileReader("file.txt"));

String line;

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

System.out.println(line);

}

reader.close();

* This code reads each line of a text file and prints it.
* BufferedReader improves reading speed by reading larger chunks.

1. **What is the purpose of file class in java and how it used for file handling?**

* The File class represents file and directory pathnames.
* It is used to check if a file exists, create new files/directories, delete files, and more.
* It does not read or write file content — it's for file-level operations.
* Example:

File file = new File("test.txt");

if (file.exists()) {

System.out.println("File exists");

}

1. **Explain how to write data to text file?**

import java.io.FileWriter;

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

writer.write("Hello, World!");

writer.close();

* This code writes the string "Hello, World!" to output.txt.
* Always close the writer to save the changes.

1. **How do you handle file exceptions in java?**

* File operations can cause exceptions like IOException, FileNotFoundException.
* These are handled using try-catch blocks.
* This program prevents file from crashing due to file errors

try {

FileWriter fw = new FileWriter("file.txt");

fw.write("Data");

fw.close();

} catch (IOException e) {

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

}

1. **What is file input stream in java?**

* FileInputStream is used to read binary data from a file.
* It reads one byte at a time or an array of bytes.
* Eg:

FileInputStream fis = new FileInputStream("data.dat");

int i = fis.read();

while(i != -1) {

System.out.print((char)i);

i = fis.read();

}

fis.close();

1. **Difference between input stream and output stream?**

|  |  |  |
| --- | --- | --- |
| **S NO** | **InputStream** | **OutputStream** |
| **1** | **Reads data from a source (e.g., file).** | **Writes data to a destination (e.g., file).** |
| **2** | **Used for input operations.** | **Used for output operations.** |
| **3** | **Common class: FileInputStream.** | **Common class: FileOutputStream.** |
| **4** | **Method: read().** | **Method: write().** |
| **5** | **Belongs to java.io package.** | **Also belongs to java.io package.** |

1. **Difference between byte stream and character stream?**

|  |  |  |
| --- | --- | --- |
| **S NO** | **Byte Stream** | **Character Stream** |
| **1** | **Handles binary data.** | **Handles text data (characters).** |
| **2** | **Uses InputStream and OutputStream.** | **Uses Reader and Writer.** |
| **3** | **Reads/writes in bytes.** | **Reads/writes in characters (Unicode).** |
| **4** | **Suitable for images, audio, video, etc.** | **Suitable for reading/writing text files.** |
| **5** | **Example: FileInputStream, FileOutputStream.** | **Example: FileReader, FileWriter.** |

1. **What are the several subclasses of input stream?**

* FileInputStream – Reads raw bytes from a file on disk.
* BufferedInputStream – Reads data into a buffer to improve read efficiency from a file.
* ByteArrayInputStream – Reads data from a byte array in memory as if it were a file.
* ObjectInputStream – Reads Java objects that were previously written using ObjectOutputStream.
* DataInputStream – Reads primitive data types (int, float, etc.) in binary format from an input stream**.**

1. **Write down the common methods of input stream?**

* read() – reads one byte of data.
* read(byte[] b) – reads bytes into an array.
* skip(long n) – skips n bytes.
* available() – returns number of available bytes to read.
* close() – closes the stream.

1. **What is random access file class in java?**

* This class allows both reading and writing at any position in the file.
* It uses a file pointer to move to a specific location.
* It supports both "r" (read-only) and "rw" (read/write) modes.

1. **What are the several subclasses of output stream?**

* FileOutputStream – Writes raw bytes directly to a file on disk.
* BufferedOutputStream – Writes data to a buffer first, then to the file for better performance.
* ObjectOutputStream – Writes Java objects to a file in a serialized (byte) format.
* DataOutputStream – Writes primitive data types (int, float, etc.) in a binary format.
* ByteArrayOutputStream – Writes output data into a byte array (in memory) instead of a file.

1. **Write down the common methods of output stream?**

* write(int b) – writes a byte.
* write(byte[] b) – writes byte array.
* flush() – flushes data to the destination.
* close() – closes the stream.

1. **What are the java file class methods?**

* createNewFile() – creates a new file.
* delete() – deletes the file.
* exists() – checks if file exists.
* getName() – returns the file name.
* mkdir() – creates a new directory.

1. **What are the several operations that we can perform on a file in java?**

* Create a new file
* Read from a file.
* Write to a file.
* Delete or rename a file.
* Check file properties (length, path, etc.).

1. **What is a scanner class in java?**

* The Scanner class is used to read input from various sources like keyboard or files.
* It belongs to java.util package.
* It provides methods like nextLine(), nextInt(), hasNextLine().
* Eg:

Scanner sc = new Scanner(new File("file.txt"));

while(sc.hasNextLine()) {

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

}

1. **How can you create a new directory in java?**

File dir = new File("newFolder");

if (dir.mkdir()) {

System.out.println("Directory created.");

}

* You can also use mkdir() to create parent directories too.

1. **What are the date classes in java?**

In Java, the main **date and time classes** are:

1. **java.util.Date**
   * A legacy class used to represent date and time.
   * Contains both date and time information but is largely replaced by newer classes.
2. **java.util.Calendar**
   * Also part of the legacy API.
   * Provides methods for date arithmetic (e.g., adding days or months), but is complex and less intuitive.
3. **java.time.LocalDate**
   * Introduced in Java 8.
   * Represents a date **without time** (e.g., 2025-08-05).
4. **java.time.LocalTime**
   * Represents a **time** without a date (e.g., 14:30:00).
5. **java.time.LocalDateTime**
   * Combines date and time, **without timezone** information.
6. **java.time.ZonedDateTime**
   * Represents date and time **with timezone**.
7. **java.time.Instant**
   * Represents a specific point in time (timestamp), often used in machine-readable formats.

These classes from java.time package (Java 8 and later) are preferred over the older Date and Calendar classes due to better design and immutability.

1. **How to display current date and time in java?**

To **display the current date and time in Java**, you can use the LocalDateTime class from the java.time package (introduced in Java 8).

Eg program:

import java.time.LocalDateTime;

public class CurrentDateTimeExample {

public static void main(String[] args) {

LocalDateTime current = LocalDateTime.now();

System.out.println("Current Date and Time: " + current);

}

}

1. **How to display local date and time in java?**

To **display local date and time in Java**, you can use the LocalDate, LocalTime, or LocalDateTime classes from the java.time package (Java 8+). These represent the date and time based on the **system's local time zone**, but **without timezone information**.

Eg program:

import java.time.LocalDate;

import java.time.LocalTime;

public class LocalDateTimeExample {

public static void main(String[] args) {

LocalDate date = LocalDate.now();

LocalTime time = LocalTime.now();

System.out.println("Local Date: " + date);

System.out.println("Local Time: " + time);

}

}

|  |
| --- |
|  |