26TH JUNE ASSIGNMENTS

1. **Explain different types of Errors in Java**

Ans: In Java, there are three types of errors that can occur:

a. Syntax Errors: These errors occur when the code violates the syntax rules of the Java programming language. These errors are caught by the compiler during the compilation phase and need to be fixed before the code can be executed.

b. Runtime Errors: Also known as exceptions, these errors occur during the execution of a program. They are not caught by the compiler but can be handled at runtime using exception handling mechanisms.

c. Logical Errors: These errors occur when the code does not produce the expected output due to a mistake in the logic or algorithm. Logical errors are not caught by the compiler or runtime system and require careful debugging to identify and fix.

2. What is an Exception in Java

Ans: Exception in Java refers to an abnormal condition or an event that occurs during the execution of a program, which disrupts the normal flow of the program. Exceptions can occur due to various reasons, such as invalid input, resource unavailability, or programming errors. When an exception occurs, an object representing the exception is created and thrown.

3. How can you handle exceptions in Java? Explain with an example

Ans: Exception handling in Java allows you to gracefully handle exceptions and provide alternative flows or error recovery mechanisms. In Java, exceptions are handled using try-catch blocks. The try block encloses the code that might throw an exception, and the catch block catches and handles the exception if it occurs. Here's an example:

```
java

Copy code

try {

    // Code that might throw an exception

    int result = divide(10, 0);

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

```
} catch (ArithmeticException e) {
    // Exception handling code
    System.out.println("Error: Division by zero");
}

public static int divide(int a, int b) {
    return a / b;
}
```

In the above example, the divide method can throw an ArithmeticException if the divisor is zero. The try block calls the divide method, and the catch block handles the exception by printing an error message.

4. Why do we need exception handling in Java?

Ans: Exception handling is necessary in Java to ensure robust and reliable software. It allows you to gracefully handle unexpected situations and recover from errors without abruptly terminating the program. Exception handling provides the following benefits:

Prevents program crashes: By catching and handling exceptions, you can prevent the program from crashing and display meaningful error messages to the user.

Facilitates error recovery: You can define alternative flows or recovery mechanisms to handle exceptions and continue program execution.

Enhances code readability: Exception handling allows you to separate error handling code from normal code, making the code more readable and maintainable.

Promotes software reliability: By handling exceptions, you can handle exceptional conditions and ensure that the software operates reliably under various circumstances.

5. What is the difference between exception and error in Java?

Ans: In Java, exceptions and errors are both subclasses of the Throwable class, but they represent different types of abnormal conditions:

Exceptions: Exceptions are used to represent exceptional conditions that can occur during the normal execution of a program. They are typically caused by errors in program logic, invalid user input, or external factors. Examples include NullPointerException, ArithmeticException, FileNotFoundException, etc. Exceptions can be caught and handled using exception handling mechanisms.

Errors: Errors, on the other hand, represent abnormal conditions that are usually beyond the control of the program. They indicate serious problems that may cause the program to terminate, such as out-of-memory errors (OutOfMemoryError), stack overflow errors (StackOverflowError), etc. Errors are generally not caught or handled in the program, as they are typically unrecoverable.

6. Name the different types of exceptions in Java

Ans: Java provides several built-in types of exceptions, including:

Checked exceptions: These are exceptions that must be declared in the method signature or handled using try-catch blocks. Examples include IOException, SQLException, ClassNotFoundException, etc.

Unchecked exceptions: These are exceptions that do not need to be declared or caught explicitly. They are subclasses of RuntimeException and its subclasses. Examples include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.

Errors: These represent severe problems that are typically beyond the control of the program, such as system errors, resource exhaustion, etc. Examples include OutOfMemoryError, StackOverflowError, NoClassDefFoundError, etc

7. Can we just use try instead of finally and catch blocks?

Ans: No, we cannot use only the try block without the catch or finally blocks. The finally block is optional, but at least one catch or finally block must be present after the try block. The catch block is used to catch and handle exceptions, while the finally block is used to specify code that should be executed regardless of whether an exception occurred or not.

8. What is the lambda expression of Java 8?

Ans: Lambda expressions were introduced in Java 8 as a concise way to represent anonymous functions. A lambda expression is a block of code that can be treated as a function and passed around as a parameter to methods or assigned to variables. It allows for functional programming constructs in Java. The syntax of a lambda expression consists of parameters, an arrow token (->), and a body. For example: (x, y) -> x + y.

9. Can you pass lambda expressions to a method? When?

Ans: Yes, lambda expressions can be passed as arguments to methods. This is possible when the method expects a functional interface as a parameter, which is an interface with a single abstract method. Since lambda expressions can be used to implement the functional interface's abstract method, they can be passed as arguments. This allows for a more concise and expressive way to define behavior at the call site. For example:

java

Copy code

```
interface Calculator {
```

```
    int calculate(int a, int b);
}

public static int performOperation(int a, int b, Calculator calculator) {
    return calculator.calculate(a, b);
}

public static void main(String[] args) {
    int result = performOperation(5, 3, (x, y) -> x + y);
    System.out.println("Result: " + result);
}
```

In the above example, the performOperation method takes a lambda expression of type Calculator as a parameter. The lambda expression (x, y) -> x + y implements the calculate method of the Calculator interface.

10. What is the functional interface in Java 8?

Ans: A functional interface in Java 8 is an interface that has exactly one abstract method. It may also contain default methods, static methods, and constant variables. Functional interfaces are used to leverage the functional programming features introduced in Java 8, such as lambda expressions and method references. They provide a way to represent behavior as a first-class citizen in Java. Examples of functional interfaces in Java 8 include Runnable, Comparator, Consumer, Function, etc.

11. Why do we use lambda expressions in Java?

Ans: Lambda expressions in Java are used to achieve functional programming constructs, such as passing behavior as arguments, writing more concise code, and enabling parallel processing. They provide a way to write more expressive and readable code by focusing on what needs to be done rather than how it should be done. Lambda expressions help in reducing boilerplate code and promoting code reusability by encapsulating behavior in a concise and flexible manner.

12. Is it mandatory for a lambda expression to have parameters?

Ans: No, it is not mandatory for a lambda expression to have parameters. Lambda expressions can be parameterless, depending on the functional interface they are implementing. For example, a functional interface with no parameters can be implemented using a lambda expression without any parameters. Here's an example:

```
java

Copy code

interface MessagePrinter {
    void printMessage();
```

```
}

public static void main(String[] args) {
    MessagePrinter printer = () -> System.out.println("Hello, World!");
    printer.printMessage();
}
```

In the above example, the MessagePrinter functional interface has a method with no parameters (printMessage). The lambda expression () -> System.out.println("Hello, World!") implements this method without any parameters.

13. What is an interface in Java?

Ans: In Java, an interface is a reference type that acts as a contract or a blueprint for classes to follow. It defines a set of methods that a class implementing the interface must implement. An interface can also include constant variables and default methods with implementations. It provides a way to achieve abstraction, multiple inheritance of type, and polymorphism in Java.

14. Which modifiers are allowed for methods in an Interface? Explain with an example

Ans: Methods in an interface can have the following modifiers:

public: Methods in an interface are implicitly public. They are accessible from other classes and interfaces.

abstract: Methods in an interface are implicitly abstract. They don't have a method body and must be implemented by the classes implementing the interface.

default: Starting from Java 8, interfaces can also have default methods. Default methods provide a default implementation for a method in the interface. They are non-abstract and can be overridden by implementing classes.

static: Starting from Java 8, interfaces can also have static methods. Static methods in interfaces are associated with the interface itself and can be called without an instance of the interface.

Here's an example:

java

Copy code

```
interface MyInterface {
```

```
    void abstractMethod(); // Implicitly public and abstract

    default void defaultMethod() {
        System.out.println("Default method");
    }

    static void staticMethod() {
        System.out.println("Static method");
    }
}
```

15. What is the use of interface in Java?Or, why do we use an interface in Java?

Ans: Interfaces in Java are used to define a contract or a set of rules that classes must adhere to. They provide a way to achieve abstraction, multiple inheritance of type, and polymorphism. Some common uses of interfaces in Java include:

Defining API contracts: Interfaces are often used to define contracts for libraries or frameworks. They specify the methods that must be implemented by the users of the library or framework.

Implementing polymorphism: Interfaces allow objects of different classes to be treated polymorphically. By programming to the interface, rather than specific implementations, you can write code that is more flexible and adaptable to different implementations.

Enforcing code consistency: Interfaces help enforce a consistent set of methods that implementing classes must provide. This promotes code organization, readability, and maintainability.

Achieving loose coupling: By programming to interfaces, you can achieve loose coupling between components of a system. This allows for easier modification, extension, and testing of code.

16. What is the difference between abstract class and interface in Java?

Ans: The main differences between an abstract class and an interface in Java are as follows:

Inheritance: A class can extend only one abstract class, but it can implement multiple interfaces.

Method implementation: An abstract class can have both abstract and non-abstract methods, including instance variables and constructors. It can provide default implementations for some or all of its methods. In contrast, an interface can only have abstract methods and constants. It

does not allow instance variables or constructors. Starting from Java 8, interfaces can have default and static methods with implementations.

Access modifiers: An abstract class can have different access modifiers (public, protected, etc.) for its members, whereas all members of an interface are implicitly public.

Instantiation: An abstract class cannot be instantiated directly using the new keyword. It needs to be subclassed and instantiated through the subclass. On the other hand, an interface cannot be instantiated at all. It defines a contract for implementing classes.

Usage: Abstract classes are typically used to provide a common base or default implementation for related classes. Interfaces, on the other hand, are used to define contracts or behaviors that multiple unrelated classes can adhere to.

In general, use an abstract class when you want to provide a common base implementation or when there is a strong "is-a" relationship between the class and its subclasses. Use an interface when you want to define a contract or behavior that multiple unrelated classes can implement.