ASSIGNMENT-5(JAVA CORE MODULE)

Q. What is Exception in Java?

Ans- In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When an exceptional condition arises, an exception is thrown, which is a way of signaling that an error or unexpected situation has occurred.

Exceptions in Java are represented by classes that inherit from the `java.lang.Exception` class or its subclasses. These exception classes provide information about the type and cause of the exception.

Java provides a mechanism for handling exceptions through the use of try-catch blocks. The try block encloses the code that may throw an exception, and the catch block(s) catch and handle the exceptions that occur. The catch block contains code to handle specific types of exceptions, allowing the program to gracefully recover from exceptional situations.

Java also supports the concept of checked and unchecked exceptions. Checked exceptions are exceptions that are checked at compile-time, and the programmer is required to handle or declare them using the `throws` keyword in the method signature. Unchecked exceptions, on the other hand, do not need to be explicitly handled or declared.

By using exceptions, developers can write code that gracefully handles errors, provides appropriate error messages, and ensures the program can recover from exceptional situations. This helps improve the robustness and reliability of Java programs.

Q2. What is Exception Handling?

Ans- Exception handling is the process of dealing with and responding to exceptions that occur during the execution of a program. It involves catching and handling exceptions to prevent the program from terminating abruptly and to provide appropriate error handling and recovery mechanisms.

In Java, exception handling is typically done using try-catch blocks. The code that is expected to throw an exception is placed within the try block. If an exception occurs within the try block, it is caught and handled by the catch block(s) that follow.

The catch block contains code that specifies the type of exception it can handle, using the `catch` keyword followed by the exception type in parentheses. When an exception of that type is thrown,

the catch block is executed, allowing the program to handle the exceptional situation in a controlled manner. Multiple catch blocks can be used to handle different types of exceptions.

Here's a basic example of exception handling in Java:

```java
try {
   // Code that may throw an exception
   // ...
} catch (ExceptionType1 e1) {
   // Handle and recover from ExceptionType1
   // ...
} catch (ExceptionType2 e2) {
   // Handle and recover from ExceptionType2
   // ...
} finally {
   // Code that always executes, regardless of whether an exception occurred or not
   // ...
}
```

In addition to try-catch blocks, Java also provides the `finally` block, which can be used to specify code that always executes, regardless of whether an exception occurred or not. The `finally` block is typically used to release resources or perform cleanup operations.

Exception handling allows programmers to gracefully handle errors and exceptional situations in their code, providing error messages, logging, and recovery mechanisms. It helps make programs more robust, reliable, and user-friendly.

Q3. What is the difference between Checked and Unchecked Exceptions and Error?

Ans- In Java, exceptions and errors are both types of exceptional conditions that can occur during the execution of a program. However, there are some important differences between checked exceptions, unchecked exceptions, and errors.

Checked Exceptions:

- Checked exceptions are exceptions that are checked by the compiler at compile-time. This means that the compiler enforces that the programmer either handles the checked exception using a try-catch block or declares that the method throws the exception using the `throws` keyword in the method signature.

- Examples of checked exceptions in Java include `IOException`, `SQLException`, and `ClassNotFoundException`.

- The purpose of checked exceptions is to ensure that exceptional conditions that can be reasonably anticipated and handled are explicitly dealt with in the code. It promotes robust error handling and forces the programmer to consider potential exceptional situations.

- If a checked exception is not caught or declared to be thrown, the code will not compile.

Unchecked Exceptions:

- Unchecked exceptions, also known as runtime exceptions, are exceptions that are not checked by the compiler at compile-time. They do not need to be explicitly handled or declared.

- Examples of unchecked exceptions in Java include `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `IllegalArgumentException`.

- Unchecked exceptions usually indicate programming errors or unexpected conditions that may arise during runtime. They often result from incorrect logic or incorrect use of APIs.

- Unchecked exceptions can be caught and handled using try-catch blocks, but they are not required to be caught or declared using the `throws` keyword.

Errors:

- Errors, like exceptions, are also exceptional conditions that can occur during the execution of a program. However, errors generally indicate serious problems that are beyond the control of the program and should not be caught or handled.

- Examples of errors in Java include `OutOfMemoryError`, `StackOverflowError`, and `NoClassDefFoundError`.

- Errors are typically caused by system-level or environment-level issues, such as insufficient resources or unrecoverable errors.

- Unlike exceptions, errors are not meant to be caught and recovered from. They indicate severe problems that usually lead to program termination.

In summary, the main differences are:

- Checked exceptions are checked by the compiler at compile-time and require explicit handling or declaration. Unchecked exceptions do not require explicit handling or declaration.

- Checked exceptions are used for anticipated and recoverable exceptional situations, while unchecked exceptions often indicate programming errors or unexpected conditions.

- Errors indicate severe problems that are usually beyond the control of the program and are not meant to be caught or handled.

Q4. What is the difference between throw and throws in Java?

Ans- In Java, `throw` and `throws` are both keywords used in exception handling, but they have different purposes:

1. `throw`:

The `throw` keyword is used to explicitly throw an exception within a program. It is used to raise an exception when a certain condition or error occurs. The `throw` statement is followed by an instance of an exception or a subclass of `Throwable` that represents the exceptional condition.

Here's an example of using the `throw` keyword:

```java
void processNumber(int number) {
   if (number < 0) {
      throw new IllegalArgumentException("Number cannot be negative");
   }
   // Process the number
}
```

In this example, if the `number` argument is negative, the `throw` statement throws an instance of `IllegalArgumentException` with a specified error message.

2. `throws`:

The `throws` keyword is used in the method declaration to specify that the method can potentially throw one or more types of exceptions. It indicates that the method may not handle the exceptions itself but rather delegates the responsibility of handling the exceptions to the calling code.

Here's an example of using the `throws` keyword:

```java
void readFile() throws IOException {
   // Code that may throw IOException
}
```

In this example, the `readFile()` method is declared with the `throws` keyword, specifying that it may throw an `IOException`. This informs the caller of the method that they need to handle the exception or propagate it further.

To handle exceptions thrown by a method declared with `throws`, the caller must either use a try-catch block to handle the exception or declare the exception using the `throws` keyword in its own method declaration.

In summary, `throw` is used to explicitly throw an exception within a method, while `throws` is used to declare that a method may throw one or more types of exceptions.

Q5. What is multithreading in java? Mention its Advantages.

Ans- Multithreading in Java refers to the concurrent execution of multiple threads within a single program. A thread is a lightweight unit of execution that can perform tasks independently and concurrently with other threads. Multithreading allows different parts of a program to run simultaneously, thus enabling concurrent and parallel execution.

Advantages of multithreading in Java include:

1. Increased responsiveness and interactivity: Multithreading allows a program to remain responsive even when performing time-consuming tasks. By executing tasks concurrently, a program can continue to handle user input and respond to events without blocking the entire execution.

2. Efficient resource utilization: Multithreading enables efficient utilization of system resources, such as CPU cycles. By dividing tasks among multiple threads, a program can make use of available resources more effectively, leading to improved performance and faster execution.

3. Enhanced program structure and modularity: Multithreading allows for the modular organization of code and separation of concerns. Different parts of a program can be encapsulated within separate threads, making the codebase more manageable, maintainable, and extensible.

4. Parallel processing and performance improvement: Multithreading facilitates parallel processing, enabling tasks to be executed simultaneously on multiple CPU cores. This can lead to significant performance improvements for computationally intensive tasks by leveraging the available hardware resources.

5. Asynchronous programming: Multithreading supports asynchronous programming models, where tasks can be executed concurrently without waiting for the completion of previous tasks. This is particularly useful for handling I/O operations, network requests, and event-driven programming, allowing for efficient utilization of system resources and better responsiveness.

6. Effective utilization of modern hardware: Multithreading is especially beneficial on systems with multiple CPU cores, where it allows programs to leverage parallelism and exploit the full potential of modern hardware architectures.

It's important to note that while multithreading offers advantages, it also introduces complexities and challenges such as thread synchronization, race conditions, and resource sharing. Careful design and consideration are required to ensure thread safety and proper synchronization when working with shared data.

Q6. Write a program in java to create and call a custom exception.

Ans-

```
class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) throws InvalidAgeException {
        if (age < 0) {
            throw new InvalidAgeException("Age cannot be negative");
        }
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
```

```
    }

    public int getAge() {
        return age;
    }
}

public class Main {
    public static void main(String[] args) {
        try {
            Person person = new Person("Amit Ranjan", -23);
            System.out.println("Name: " + person.getName());
            System.out.println("Age: " + person.getAge());
        } catch (InvalidAgeException e) {
            System.out.println("Invalid Age: " + e.getMessage());
        }
    }
}
```

In this example, we create a custom exception called `InvalidAgeException` by extending the base `Exception` class. The `InvalidAgeException` class has a constructor that accepts a message parameter, which is passed to the superclass constructor using the `super` keyword.

Next, we define a `Person` class that takes a name and age as parameters in its constructor. Within the `Person` constructor, we check if the provided age is negative. If it is, we throw an instance of the `InvalidAgeException` with a specific error message.

In the `Main` class, we attempt to create a `Person` object with a negative age value. Since the age is invalid, an `InvalidAgeException` is thrown. We catch the exception using a try-catch block and handle it by displaying the error message.

Note that the `InvalidAgeException` is a checked exception, so it needs to be either caught or declared to be thrown in the `Person` constructor using the `throws` keyword.

Q7.How can you handle exceptions in java?

Ans- In Java, exceptions can be handled using try-catch blocks and/or the `throws` keyword. Here are the different ways to handle exceptions in Java:

1. Using try-catch blocks:

```java
try {
    // Code that may throw an exception
} catch (ExceptionType1 e1) {
    // Handle and recover from ExceptionType1
} catch (ExceptionType2 e2) {
    // Handle and recover from ExceptionType2
} finally {
    // Code that always executes, regardless of whether an exception occurred or not
}
```

- The code that may throw an exception is placed within the `try` block.
- If an exception occurs within the `try` block, it is caught and handled by the appropriate `catch` block(s) that follow.
- Multiple `catch` blocks can be used to handle different types of exceptions.
- The `finally` block, if present, contains code that always executes, regardless of whether an exception occurred or not. It is typically used for releasing resources or performing cleanup operations.

2. Declaring exceptions with the `throws` keyword:

```java
void method() throws ExceptionType1, ExceptionType2 {
    // Code that may throw exceptions
}
```

- The `throws` keyword is used in the method declaration to specify that the method may throw one or more types of exceptions.
- It indicates that the method does not handle the exceptions itself but delegates the responsibility of handling them to the calling code.
- The caller of the method must either handle the exceptions using try-catch blocks or declare the exceptions to be thrown using the `throws` keyword in its own method declaration.

3. Using a combination of try-catch blocks and throws:

```java
void method() throws ExceptionType1, ExceptionType2 {
    try {
        // Code that may throw exceptions
    } catch (ExceptionType1 e1) {
        // Handle and recover from ExceptionType1
    } catch (ExceptionType2 e2) {
        // Handle and recover from ExceptionType2
    }
}
```

- In this approach, the method declares the exceptions it can throw using the `throws` keyword.
- Inside the method, the exceptions are caught and handled using try-catch blocks.

When handling exceptions, it is important to consider proper error handling, such as displaying error messages, logging, or taking appropriate recovery actions. It helps in identifying and resolving issues, making programs more robust and reliable.

Q8. What is Thread in Java?

Ans- In Java, a thread refers to a separate path of execution within a program. It is a lightweight unit of execution that can perform tasks concurrently with other threads. Threads allow different parts of a program to run independently and simultaneously, enabling concurrent and parallel execution.

In Java, threads are implemented using the `Thread` class or by implementing the `Runnable` interface. The `Thread` class provides built-in methods and features for managing threads, while implementing the `Runnable` interface allows objects to be executed as threads.

Here are some key concepts related to threads in Java:

1. Multithreading: Multithreading refers to the execution of multiple threads within a single program. By dividing tasks among multiple threads, a program can achieve concurrent execution, enabling different parts of the program to run simultaneously.

2. Main Thread: When a Java program starts, it automatically creates a main thread, which serves as the entry point of the program. The main thread executes the `main()` method and can create additional threads as needed.

3. Thread States: Threads can exist in different states during their lifecycle, including New, Runnable, Blocked, Waiting, Timed Waiting, and Terminated. These states reflect the different stages of a thread's execution, such as creation, running, waiting, and completion.

4. Synchronization: In multithreaded environments, it's important to synchronize access to shared resources to avoid conflicts and race conditions. Java provides synchronization mechanisms such as the `synchronized` keyword, locks, and monitors to ensure thread safety.

5. Thread Priorities: Threads can be assigned different priorities ranging from 1 to 10, where a higher priority indicates a higher preference for execution. Thread priorities can influence the order and frequency of thread execution, although the actual behavior depends on the underlying operating system.

6. Thread Scheduling: Thread scheduling is the process of determining which thread should execute when multiple threads are ready to run. The thread scheduler, provided by the operating system, determines the order in which threads are executed based on factors like priority and fairness.

Threads are commonly used in scenarios such as parallel processing, concurrent programming, handling user interfaces, network communication, and background tasks. However, working with threads requires careful consideration and proper synchronization to avoid issues like race conditions and thread interference.

Q9. What are the two ways of implementing threads in java?

Ans- In Java, there are two main ways to implement threads: by extending the `Thread` class or by implementing the `Runnable` interface.

1. Extending the `Thread` class:

- In this approach, you create a new class that extends the `Thread` class and override its `run()` method, which contains the code that will be executed in the new thread.

- Here's an example:

```java
class MyThread extends Thread {
    public void run() {
        // Code to be executed in the new thread
```

```
    }
}
```

- To start the new thread, you create an instance of your `MyThread` class and call its `start()` method:

```java
MyThread myThread = new MyThread();
myThread.start();
```

2. Implementing the `Runnable` interface:

- In this approach, you create a new class that implements the `Runnable` interface and implement its `run()` method, which contains the code to be executed in the new thread.

- Here's an example:

```java
class MyRunnable implements Runnable {
    public void run() {
        // Code to be executed in the new thread
    }
}
```

- To start the new thread, you create an instance of your `MyRunnable` class and pass it to a `Thread` object's constructor. Then, you call the `start()` method on the `Thread` object:

```java
MyRunnable myRunnable = new MyRunnable();
Thread thread = new Thread(myRunnable);
thread.start();
```

Both approaches allow you to define the code to be executed in a separate thread. The key difference is that when extending the `Thread` class, the class itself represents the thread, while when implementing the `Runnable` interface, you pass an instance of the class to a `Thread` object.

Implementing the `Runnable` interface is generally preferred over extending the `Thread` class because it offers better flexibility and separation of concerns. It allows you to separate the thread's behaviour from the thread management, as you can reuse the same `Runnable` instance with multiple threads or use it with thread pools.

Q10. What do you mean by Garbage collection?

Ans- Garbage collection is an automatic memory management mechanism in programming languages like Java. It is the process of automatically reclaiming memory occupied by objects that are no longer in use, freeing up resources and preventing memory leaks.

In languages with garbage collection, such as Java, memory allocation and deallocation are handled by the runtime environment. The programmer does not need to explicitly allocate or deallocate memory for objects. Instead, the garbage collector identifies and collects objects that are no longer reachable or referenced by the program.

The garbage collector performs the following main tasks:

1. Marking: The garbage collector starts by identifying all objects that are reachable from the root of the object graph, typically starting with the main execution thread and static variables. It marks these objects as live.

2. Tracing: The garbage collector traces object references starting from the marked live objects and follows references to other objects, marking them as live as well. This process continues recursively until all reachable objects are marked.

3. Sweep and Free: Once all reachable objects are marked as live, the garbage collector sweeps through the memory space, identifying and freeing memory occupied by objects that are not marked as live. This memory is then made available for future allocations.

Java's garbage collector operates automatically in the background, periodically reclaiming memory and managing the heap space. The exact behavior and algorithms used by the garbage collector may vary across different Java implementations.

Benefits of garbage collection include:

- Simplicity: Garbage collection simplifies memory management by automating the process of memory deallocation. Developers can focus more on writing application logic rather than managing memory explicitly.

- Memory Leak Prevention: Garbage collection helps prevent memory leaks, which occur when memory is allocated but not properly deallocated, leading to excessive memory usage and potential program instability. The garbage collector ensures that objects no longer in use are automatically freed.

- Dynamic Memory Management: Garbage collection allows for dynamic memory allocation, where objects can be created and destroyed as needed without the programmer having to manually manage memory.

- Improved Application Performance: Garbage collection can help optimize memory usage and improve overall application performance by efficiently reclaiming memory and reducing memory fragmentation.

However, it's important to note that garbage collection is not without its challenges. The process of garbage collection can introduce pauses or "stop-the-world" events in the application, where the program execution is temporarily halted while the garbage collector runs. Careful design and tuning of the garbage collector settings may be required to minimize these pauses and optimize application performance.
