**Assignment Questions 5**

<aside> 💡 **Q1.What is Exception in Java?**

</aside>

<aside> 💡 **Q2.What is Exception Handling?**

</aside>

<aside> 💡 **Q3.What is the difference between Checked and Unchecked Exceptions and Error?**

</aside>

<aside> 💡 **Q4.What are the difference between throw and throws in Java?**

</aside>

<aside> 💡 **Q5.What is multithreading in Java? mention its advantages**

</aside>

<aside> 💡 **Q6.Write a program to create and call a custom exception**

</aside>

<aside> 💡 **Q7.How can you handle exceptions in Java?**

</aside>

<aside> 💡 **Q8.What is Thread in Java?**

</aside>

<aside> 💡 **Q9. What are the two ways of implementing thread in Java?**

</aside>

<aside> 💡 **Q10.What do you mean by garbage collection?**

</aside>

Q1. What is an Exception in Java?

In Java, an exception is an event that occurs during the execution of a program, disrupting the normal flow of the program. It represents an abnormal condition or error situation that needs to be handled. Exceptions allow programs to handle errors and exceptional situations in a structured and controlled manner.

Q2. What is Exception Handling?

Exception handling is the process of dealing with exceptions that occur during the execution of a program. It involves identifying, catching, and handling exceptions to prevent the program from terminating abnormally. Exception handling allows for graceful recovery from exceptional conditions and helps in making programs more robust and reliable.

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

* Checked Exceptions: Checked exceptions are exceptions that are checked at compile-time. The compiler enforces that the code either handles the checked exception using a **try-catch** block or declares it using the **throws** clause. Checked exceptions typically represent recoverable conditions that the program can handle.
* Unchecked Exceptions (Runtime Exceptions): Unchecked exceptions are exceptions that are not checked at compile-time. The compiler does not enforce handling or declaration of unchecked exceptions. Unchecked exceptions typically represent programming errors or unexpected conditions that are not recoverable.
* Error: Errors are a separate category of problems that are not meant to be caught or handled by normal application code. Errors represent severe problems that are typically beyond the control of the application, such as out-of-memory errors or system failures. Errors are usually unrecoverable and are not meant to be caught or handled by application-level exception handling mechanisms.

Q4. What are the differences between throw and throws in Java?

* **throw** is used to explicitly throw an exception within a method or block of code. It is followed by an instance of an exception class or a subclass of **Throwable**. When **throw** is encountered, it immediately stops the execution of the current block of code and transfers control to the nearest matching **catch** block or to an exception handler higher up in the call stack.
* **throws** is used in a method declaration to indicate that the method may throw one or more types of exceptions. It specifies the exceptions that the method might throw, but it does not actually throw the exceptions itself. By declaring the exceptions using **throws**, it informs the caller of the method that they need to handle or propagate the exceptions.

Q5. What is multithreading in Java? Mention its advantages.

Multithreading in Java refers to the concurrent execution of multiple threads within a single program. A thread is a lightweight unit of execution that represents a separate path of execution in a program.

Advantages of multithreading in Java include improved performance, enhanced responsiveness, concurrency and parallelism, resource sharing, background processing, modularity and maintainability, real-time application support, utilization of multi-core processors, and asynchronous programming capabilities.

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

javaCopy code

class MyCustomException extends Exception { public MyCustomException(String message) { super(message); } } public class CustomExceptionExample { public static void main(String[] args) { try { throw new MyCustomException("This is a custom exception"); } catch (MyCustomException e) { System.out.println("Caught custom exception: " + e.getMessage()); } } }

In this example, we create a custom exception class **MyCustomException** that extends the **Exception** class. We provide a constructor that accepts a message to be passed to the parent **Exception** class.

In the **main** method, we deliberately throw an instance of **MyCustomException** using the **throw** statement. We catch the exception using a **catch** block specific to **MyCustomException** and print the exception message.

Q7. How can you handle exceptions in Java?

Exceptions in Java can be handled using the **try-catch** block. The **try** block contains the code that might throw an exception, and the **catch** block catches and handles the exception.

Here's an example:

javaCopy code

try { // Code that might throw an exception } catch (ExceptionType1 exception1) { // Code to handle exception1 } catch (ExceptionType2 exception2) { // Code to handle exception2 } finally { // Optional: Code that is always executed, regardless of whether an exception occurs or not }

When an exception occurs within the **try** block, the code execution is immediately transferred to the appropriate **catch** block that matches the exception type. The **catch** block contains the code to handle the exception.

The **finally** block is optional and is used to specify code that should be executed regardless of whether an exception occurs or not. It is commonly used to release resources or perform cleanup operations.

Q8. What is a Thread in Java?

In Java, a thread is a separate path of execution that can run concurrently with other threads within a program. Threads allow multiple tasks to be executed simultaneously, leading to better utilization of system resources and improved performance.

Threads can be created and managed in Java using the **Thread** class or by implementing the **Runnable** interface. Each thread represents an independent flow of control within a program.

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

In Java, there are two ways to implement threads:

1. By extending the **Thread** class: You can create a new class that extends the **Thread** class and override the **run()** method. The **run()** method contains the code that will be executed when the thread is started. The **Thread** class provides other methods for thread management, such as starting and stopping threads.
2. By implementing the **Runnable** interface: You can create a class that implements the **Runnable** interface and implements the **run()** method. The **run()** method contains the code that will be executed when the thread is started. The **Runnable** interface is preferred when you want to separate the thread's behavior from the class hierarchy, as it allows for better code organization and flexibility.

Q10. What do you mean by garbage collection?

Garbage collection in Java is the process of automatically reclaiming memory occupied by objects that are no longer in use by the program. When an object is created in Java, memory is allocated to store the object. Over time, as the program executes, objects may become unreachable or no longer referenced by any part of the program.

The garbage collector in Java identifies such objects and frees the memory occupied by them, making it available for reuse by the program. Garbage collection helps in managing memory efficiently and automatically, relieving the programmer from explicitly deallocating memory.

The Java Virtual Machine (JVM) is responsible for performing garbage collection. The garbage collector runs in the background, periodically reclaiming memory from unused objects. The exact algorithms and strategies used for garbage collection may vary between different JVM implementations.

Garbage collection provides several benefits, including automatic memory management, prevention of memory leaks, reduced risk of memory-related errors, and improved developer productivity by eliminating the need for manual memory deallocation.