JAVA EXCEPTION HANDLING INTERVIEW QUESTIONS

Contents

What are the differences between Checked Exception and Unchecked Exception?

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. There are two main types of exceptions: checked exceptions and unchecked exceptions.

Checked exceptions are exceptions that are checked at compile time. This means that if a method declares that it throws a checked exception, then the code that calls the method must either handle the exception or declare that it throws the exception. Checked exceptions include exceptions that are defined in the java.lang package, such as IOException and SQLException, as well as any user-defined exceptions that extend the Exception class.

Unchecked exceptions, on the other hand, are exceptions that are not checked at compile time. This means that code that throws an unchecked exception does not need to declare or handle the exception. Unchecked exceptions include exceptions that are defined in the java.lang package, such as RuntimeException and Error, as well as any user-defined exceptions that extend the RuntimeException class.

The main difference between checked and unchecked exceptions is the way they are handled by the compiler. Checked exceptions are required to be declared or handled, while unchecked exceptions are not. This means that checked exceptions are more suitable for exceptions that are recoverable and can be handled by the calling code, while unchecked exceptions are more suitable for situations where recovery is not possible or the exception is unlikely to occur.

Overall, the choice of whether to use a checked or unchecked exception will depend on the specific requirements of your program and the type of exception that you are dealing with.

What is the difference between Exception and Error in java?

In Java, an exception is an event that occurs during the execution of a program that disrupts the normal flow of instructions. An error is a serious problem that occurs during the execution of a program and cannot be handled by the program itself.

There are two main types of exceptions in Java: checked exceptions and unchecked exceptions. Checked exceptions are exceptions that are checked at compile time and must be declared or handled by the calling code. Unchecked exceptions are exceptions that are not checked at compile time and do not need to be declared or handled.

Errors, on the other hand, are not exceptions at all. They are subclasses of the Error class, which is a subclass of Throwable. Unlike exceptions, errors are not intended to be caught or handled by the program. Instead, they indicate serious problems that cannot be handled by the program and require action by the user or the system.

Some examples of errors in Java include:

OutOfMemoryError: Indicates that the JVM has run out of memory and cannot allocate any more.

StackOverflowError: Indicates that the program has exceeded the maximum depth of the call stack.

NoClassDefFoundError: Indicates that the JVM cannot find the definition of a class that is referenced by the program.

Overall, the main difference between exceptions and errors in Java is the way they are intended to be handled.

Here is a sample Java program that demonstrates how to throw an OutOfMemoryError:

```java
public class OutOfMemoryErrorDemo {
  public static void main(String[] args) {
    // Allocate an array of very large size
    long[] array = new long[Integer.MAX_VALUE];
  }
}
```

In this example, we try to allocate an array with a size that is larger than the maximum allowed by the JVM. This will cause the JVM to run out of memory and throw an OutOfMemoryError.

To see this error in action, you can compile and run the program using the java command. For example:

```
javac OutOfMemoryErrorDemo.java

java OutOfMemoryErrorDemo
```

When the program is run, the JVM will attempt to allocate the large array, but it will not have enough memory to do so. This will cause an OutOfMemoryError to be thrown, which will be printed to the console.

Note that you may need to increase the maximum heap size of the JVM using the -Xmx command-line option in order to reproduce this error.

Here is a sample Java program that demonstrates how to throw a StackOverflowError:

```java
public class StackOverflowErrorDemo {
  public static void main(String[] args) {
    // Call a method that recursively calls itself
    infiniteRecursion();
  }

  public static void infiniteRecursion() {
    infiniteRecursion();
  }
}
```

In this example, we define a method called infiniteRecursion() that calls itself indefinitely. This will cause the call stack to grow without bound, eventually leading to a StackOverflowError.

To see this error in action, you can compile and run the program using the java command. For example:

```
javac StackOverflowErrorDemo.java
```

```
java StackOverflowErrorDemo
```

When the program is run, the infiniteRecursion() method will be called and will continue to call itself indefinitely. This will cause the call stack to grow until it exceeds the maximum allowed size, at which point a StackOverflowError will be thrown.

Note that you may need to increase the maximum stack size of the JVM using the -Xss command-line option in order to reproduce this error.

What is the difference between throw and throws?

In Java, throw is a keyword that is used to throw an exception explicitly. It is used to signal that an exceptional condition has occurred and that the normal flow of control should be interrupted.

For example, consider the following code:

```
public void divide(int x, int y) throws ArithmeticException {
    if (y == 0) {
      throw new ArithmeticException("Division by zero");
    }
    int result = x / y;
    // ...
  }
```

In this example, the divide() method throws an ArithmeticException if the value of y is zero. This exception is thrown using the throw keyword.

On the other hand, throws is a keyword that is used in the declaration of a method to indicate that the method may throw one or more exceptions. It is used to inform the calling code that it may need to handle the exceptions that are thrown by the method.

For example, consider the following code:

```
public void divide(int x, int y) throws ArithmeticException {
    // ...
  }
```

In this example, the divide() method declares that it may throw an ArithmeticException. This means that the calling code must either handle the exception or declare that it throws the exception.

What is the importance of finally block in exception handling?

In Java, the finally block is a block of code that is used in conjunction with a try-catch block to provide a way to execute code regardless of whether an exception is thrown or caught. It is typically used to perform cleanup tasks, such as closing resources or releasing resources.

The finally block is optional, but it is generally recommended to use it whenever you are using a try-catch block. This is because the finally block provides a guarantee that the code within it will be executed, regardless of whether an exception is thrown or caught.

Here is an example of how the finally block can be used:

```java
try {
  // Code that may throw an exception
} catch (SomeException e) {
  // Code to handle the exception
} finally {
  // Cleanup code that is always executed
}
```

In this example, the code in the try block may throw an exception, which will be caught by the catch block if it occurs. Regardless of whether an exception is thrown or caught, the code in the finally block will be executed.

The importance of the finally block lies in the fact that it provides a way to ensure that certain code is always executed, even if an exception is thrown.

What will happen to the Exception object after exception handling?

In Java, an exception object is created when an exception is thrown. If the exception is not caught and handled by the program, it will propagate up the call stack until it is caught by a try-catch block or until it reaches the top of the stack, at which point the program will terminate with an error.

If the exception is caught and handled by a try-catch block, the exception object will be passed to the catch block as an argument. The catch block can then use the exception object to get information about the exception, such as the type of exception, the message, and the stack trace.

After the exception has been caught and handled by the catch block, the exception object is no longer needed. It is up to the catch block to decide what to do with the exception object. In most cases, the exception object will simply be discarded and the program will continue executing.

It is important to note that the exception object is not the same thing as the exception itself. The exception object is simply an instance of a class that represents the exception and contains information about it. The exception itself is the event that occurs during the execution of the program that disrupts the normal flow of instructions. Once the exception has been caught and handled, the event itself is considered to be over and the program can continue executing.

What purpose does the keywords final, finally, and finalize fulfill?

In Java, the keywords final, finally, and finalize have different purposes.

The keyword final is used to indicate that a variable, method, or class cannot be modified or overridden. When a variable is declared as final, it cannot be reassigned to a different value. When a method is declared as final, it cannot be overridden by a subclass. When a class is declared as final, it cannot be subclassed.

The keyword finally is used in conjunction with a try-catch block to provide a way to execute code regardless of whether an exception is thrown or caught. It is typically used to perform cleanup tasks, such as closing resources or releasing resources.

The finalize() method is a method that is defined in the Object class and can be overridden by subclasses. It is called by the garbage collector when an object is about to be reclaimed, and it gives the object an opportunity to perform any necessary cleanup tasks before it is discarded.

Overall, the keyword final is used to indicate that something cannot be modified, the keyword finally is used to provide a way to execute code regardless of whether an exception is thrown or caught, and the finalize() method is used to perform cleanup tasks when an object is about to be reclaimed by the garbage collector.