EXCEPTION HANDLING

OBJECT ORIENTED PROGRAMMING I

Sercan Külcü | Object Oriented Programming I | 10.01.2023

Contents

Introduction

Java is a popular object-oriented programming language that is widely used in developing robust and scalable applications. One of the critical features that make Java stand out is its robust exception handling mechanism. In this chapter, we will delve into the concept of exception handling in Java, including what it is, why it is essential, how it works, and best practices for implementing it in your code.

What is Exception Handling?

Exception handling is the process of identifying, anticipating, and handling errors that occur during the execution of a program. In Java, an exception is an event that interrupts the normal flow of program execution, causing the program to terminate abnormally. Exceptions can occur due to various reasons, including invalid inputs, runtime errors, logical errors, and system failures.

An exception is an event that occurs during the execution of a Java program that disrupts the normal flow of the program. Exceptions can be caused by a variety of factors, such as invalid input, errors in the program, or problems with the hardware or operating system.

When an exception occurs, the Java Virtual Machine (JVM) will throw an exception object. This object contains information about the exception, such as the type of exception, the line number where the exception occurred, and the stack trace.

The program can then catch the exception and handle it in a variety of ways. The most common way to handle an exception is to print a message to the console or to display an error message to the user. The program can also try to recover from the exception by taking corrective action.

Why is Exception Handling Essential?

Exception handling is an essential feature of Java because it enables developers to handle errors in a structured and systematic way. Without exception handling, programs would crash whenever an error occurs, making it challenging to locate and fix the problem. With exception handling, developers can handle errors gracefully, allowing the program to recover from errors and continue executing.

Major reasons why an exception occurs

There are many reasons why an exception can occur in Java. Some of the most common reasons include:

- Invalid input. This can happen when the user enters invalid data, such as a non-numeric value for an integer field.
- Errors in the program. This can happen when the programmer makes a mistake in the code, such as dividing by zero or trying to access a non-existent object.
- Problems with the hardware or operating system. This can happen when there is a hardware failure, such as a memory error, or when the operating system is not functioning properly.

When an exception occurs, the Java Virtual Machine (JVM) will throw an exception object. This object contains information about the exception, such as the type of exception, the line number where the exception occurred, and the stack trace.

The program can then catch the exception and handle it in a variety of ways. The most common way to handle an exception is to print a message to the console or to display an error message to the user. The program can also try to recover from the exception by taking corrective action.

It is important to handle exceptions properly in order to ensure that your Java programs are robust and can handle unexpected events. By following the best practices for exception handling, you can help to prevent your programs from crashing or producing unexpected results.

Here are some tips for handling exceptions in Java:

- Always use try-catch blocks to handle exceptions. This will help to prevent your programs from crashing.
- Catch the specific exception type whenever possible. This will allow you to handle the exception more effectively.
- Use the finally block to close resources, even if an exception occurs. This will help to prevent resource leaks.
- Log exceptions to a file. This will help you to track down and fix errors in your programs.

By following these tips, you can help to ensure that your Java programs are robust and can handle unexpected events.

Types of Exceptions

There are two main types of exceptions in Java: checked exceptions and unchecked exceptions.

Checked exceptions are exceptions that are considered to be recoverable. These exceptions are typically caused by errors in the program, such as invalid input or errors in the logic. Checked exceptions must be declared in the method signature.

Unchecked exceptions are exceptions that are considered to be unrecoverable. These exceptions are typically caused by problems with the hardware or operating system, such as a network error or a disk full error. Unchecked exceptions do not need to be declared in the method signature.

Exception Hierarchy in java

The Java exception hierarchy is a tree-like structure that represents the relationship between different types of exceptions. The root of the hierarchy is the Throwable class, which is the parent class of all exceptions in Java. The Throwable class has two direct subclasses: Error and Exception.

The Error class represents serious errors that are usually caused by problems with the Java Virtual Machine (JVM) or the operating system. These errors are typically unrecoverable and should not be handled by the application code. Some examples of errors include:

- OutOfMemoryError
- StackOverflowError
- AssertionError

The Exception class represents all other types of exceptions. These exceptions can be caused by a variety of factors, such as invalid input, errors in the program, or problems with the hardware or operating system. Some examples of exceptions include:

- NullPointerException
- NumberFormatException
- ClassNotFoundException

The Exception class has many subclasses, each representing a specific type of exception. For example, the IOException class represents exceptions that occur when reading or writing to a file. The SQLException class represents exceptions that occur when accessing a database.

The exception hierarchy can be helpful for programmers to understand the different types of exceptions that can occur in Java. By understanding the exception hierarchy, programmers can write more robust code that can handle unexpected events.

How Does Exception Handling Work?

In Java, exception handling involves three main components: try, catch, and finally blocks. The try block contains the code that is susceptible to errors, while the catch block contains the code that handles the exception. The finally block contains code that executes regardless of whether an exception occurs or not.

Here is an example of how exception handling works in Java:

```
try {
    // code that may cause an exception
}
catch (ExceptionType exceptionObject) {
    // code to handle the exception
}
finally {
    // code to execute regardless of whether an
exception occurs or not
}
```

In this example, the code in the try block is executed first. If an exception occurs, the catch block is executed, which handles the exception. If no exception occurs, the catch block is skipped, and the finally block is executed.

How Does JVM handle an Exception?

When an exception occurs in Java, the Java Virtual Machine (JVM) will handle the exception in the following steps:

The JVM will create an exception object. The exception object will contain information about the exception, such as the type of exception, the line number where the exception occurred, and the stack trace.

The JVM will search the call stack for a matching exception handler. A matching exception handler is a block of code that is defined using the try-catch statement and that can handle the exception that was thrown.

If the JVM finds a matching exception handler, the exception handler will be executed. The exception handler can be used to print a message to the console, display an error message to the user, or try to recover from the exception.

If the JVM does not find a matching exception handler, the JVM will terminate the program.

Java Exception Keywords

There are five keywords in Java that are used to handle exceptions:

- try
- catch
- throw
- throws
- finally

The try keyword is used to define a block of code that is potentially dangerous. If an exception occurs in the try block, the catch block will be executed.

The catch keyword is used to define a block of code that will handle an exception. The catch block can be used to print a message to the console, display an error message to the user, or try to recover from the exception.

The throw keyword is used to explicitly throw an exception. It can be used for both checked and unchecked exceptions.

The throws keyword is used to declare an exception. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

The finally keyword is used to execute code regardless of whether an exception occurs. The finally statement is typically used to close resources, such as files or database connections.

The try-catch Statement

The try-catch statement is used to handle exceptions in Java. The try-catch statement has two parts: the try block and the catch block.

The try block contains the code that is potentially dangerous. If an exception occurs in the try block, the catch block will be executed.

The catch block contains the code that will handle the exception. The catch block can be used to print a message to the console, display an error message to the user, or try to recover from the exception.

The finally Statement

The finally statement is used to execute code regardless of whether an exception occurs. The finally statement is typically used to close resources, such as files or database connections.

The finally statement is always executed, even if an exception occurs. This is because the finally statement is executed after the try-catch block has finished executing.

Best Practices for Implementing Exception Handling

To ensure that your Java programs are robust and error-free, it is essential to follow best practices for implementing exception handling. Here are some tips to keep in mind:

Use specific exceptions: Always use specific exceptions rather than generic ones to catch errors. This helps in identifying the error and debugging the program quickly.

Handle exceptions gracefully: Always handle exceptions gracefully, even if they are unexpected. This helps in maintaining the stability of the program and preventing it from crashing.

Avoid catching runtime exceptions: Runtime exceptions are unchecked exceptions that occur at runtime and can be challenging to predict. Avoid catching these exceptions unless necessary.

Log exceptions: Always log exceptions when they occur to aid in debugging the program.

IllegalArgumentException

IllegalArgumentException is a subclass of RuntimeException, which means it is an unchecked exception. Unchecked exceptions are not required to be declared in the throws clause of a method.

IllegalArgumentException is thrown when a method is passed an argument that is not valid for that method. For example, if a method expects an integer argument, and a string argument is passed, an IllegalArgumentException will be thrown.

Here is an example of how IllegalArgumentException can be thrown:

```
public void divide(int x, int y) {

    if (y == 0) {

        throw new IllegalArgumentException("Division by zero");

    }

    int z = x / y;

}
```

In this example, the divide() method will throw an IllegalArgumentException if the y argument is 0.

IllegalArgumentException can be handled using a try-catch block. The following code shows how to handle an IllegalArgumentException:

```
public void divide(int x, int y) {

    try {

        int z = x / y;

    } catch (IllegalArgumentException e) {

        System.out.println("Division by zero");

    }

}
```

In this example, the divide() method will print a message to the console if the y argument is 0.

Here are some tips for avoiding IllegalArgumentException:

- Check the arguments passed to your methods before you use them.
- Use the appropriate data types for your arguments.
- Document the expected arguments for your methods.

By following these tips, you can help to avoid IllegalArgumentException in your Java code.

ArrayOutOfBoundException

ArrayIndexOutOfBoundsException is a subclass of IndexOutOfBoundsException, which means it is an unchecked exception. Unchecked exceptions are not required to be declared in the throws clause of a method.

ArrayIndexOutOfBoundsException is thrown when an attempt is made to access an element of an array that is outside the bounds of the array. For example, if an array has 10 elements, and an attempt is made to access the 11th element, an ArrayIndexOutOfBoundsException will be thrown.

Here is an example of how ArrayIndexOutOfBoundsException can be thrown:

```
int[] arr = new int[10];


arr[11] = 10; // This will throw an ArrayIndexOutOfBoundsException
```

In this example, an attempt is made to access the 11th element of the array, which is outside the bounds of the array.

ArrayIndexOutOfBoundsException can be handled using a try-catch block. The following code shows how to handle an ArrayIndexOutOfBoundsException:

```
int[] arr = new int[10];


try {

    arr[11] = 10;

} catch (ArrayIndexOutOfBoundsException e) {

    System.out.println("ArrayIndexOutOfBoundsException");

}
```

In this example, the code will print a message to the console if an attempt is made to access an element of the array that is outside the bounds of the array.

Here are some tips for avoiding ArrayIndexOutOfBoundsException:

- Make sure that you are accessing elements of an array within the bounds of the array.
- Use a for loop to iterate over the elements of an array.
- Use the length property of an array to determine the number of elements in the array.

By following these tips, you can help to avoid ArrayIndexOutOfBoundsException in your Java code.

StackOverflowException

StackOverflowException is a subclass of RuntimeException, which means it is an unchecked exception. Unchecked exceptions are not required to be declared in the throws clause of a method.

StackOverflowException is thrown when a method calls itself recursively too many times. For example, if a method calls itself recursively 100 times, a StackOverflowException will be thrown.

Here is an example of how StackOverflowException can be thrown:

```
public void recursive() {

    recursive(); // This will throw a StackOverflowException

}
```

In this example, the recursive() method calls itself recursively, which will eventually lead to a StackOverflowException.

StackOverflowException can be handled using a try-catch block. The following code shows how to handle a StackOverflowException:

```
public void recursive() {

    try {

        recursive();

    } catch (StackOverflowException e) {

        System.out.println("StackOverflowException");

    }

}
```

In this example, the code will print a message to the console if a StackOverflowException is thrown.

Here are some tips for avoiding StackOverflowException:

- Avoid using recursive methods.
- If you must use recursive methods, make sure that they are not called recursively too many times.
- Use a for loop or while loop instead of a recursive method to iterate over a collection.

By following these tips, you can help to avoid StackOverflowException in your Java code.

NumberFormatException

NumberFormatException is a subclass of RuntimeException, which means it is an unchecked exception. Unchecked exceptions are not required to be declared in the throws clause of a method.

NumberFormatException is thrown when an attempt is made to convert a string to a numeric value, but the string does not have the appropriate format. For example, if a string is "123abc", it cannot be converted to an integer, and a NumberFormatException will be thrown.

Here is an example of how NumberFormatException can be thrown:

```
// This will throw a NumberFormatException

int x = Integer.parseInt("123abc");
```

In this example, an attempt is made to convert the string "123abc" to an integer, but the string does not have the appropriate format.

NumberFormatException can be handled using a try-catch block. The following code shows how to handle a NumberFormatException:

```
int x;

try {

    x = Integer.parseInt("123abc");

} catch (NumberFormatException e) {

    System.out.println("Invalid input");

}
```

In this example, the code will print a message to the console if an attempt is made to convert a string to a numeric value, but the string does not have the appropriate format.

Here are some tips for avoiding NumberFormatException:

- Make sure that the strings that you are trying to convert to numeric values are in the correct format.
- Use the Integer.parseInt() method to convert strings to integers.
- Use the Double.parseDouble() method to convert strings to doubles.

By following these tips, you can help to avoid NumberFormatException in your Java code.

NullPointerException

NullPointerException is a subclass of RuntimeException, which means it is an unchecked exception. Unchecked exceptions are not required to be declared in the throws clause of a method.

NullPointerException is thrown when an attempt is made to dereference a null reference. For example, if a variable is declared as a reference type, but the variable is not initialized, an attempt to dereference the variable will result in a NullPointerException.

Here is an example of how NullPointerException can be thrown:

```
String str = null;

// This will throw a NullPointerException

System.out.println(str.length());
```

In this example, an attempt is made to dereference the null reference str.

NullPointerException can be handled using a try-catch block. The following code shows how to handle a NullPointerException:

```
String str;

try {

    System.out.println(str.length());

} catch (NullPointerException e) {

    System.out.println("The string is null");

}
```

In this example, the code will print a message to the console if an attempt is made to dereference a null reference.

Here are some tips for avoiding NullPointerException:

- Make sure that all references are initialized before they are used.

- Use the == operator to check if a reference is null before you dereference it.
- Use the null pointer assertion operator to check if a reference is null before you dereference it.

By following these tips, you can help to avoid NullPointerException in your Java code.

Conclusion

In summary, exception handling is a crucial feature in Java that enables developers to handle errors in a structured and systematic way. With the try, catch, and finally blocks, developers can gracefully handle exceptions and prevent programs from crashing. By following best practices for implementing exception handling, developers can ensure that their Java programs are robust, error-free, and maintainable.