COURSEWARE

Exceptions

Contents

Overview

An exception is an abnormal condition; in Java it is an event that disrupts the normal flow of a program.
An object is thrown at runtime for this, the object is contextual to the type of exception.
Every system is going to have exceptions that occur at some point, exception handling is the way in which we prevent the system from terminating unexpectedly and it is all about handling the errors in a desirable way.
Exceptions are not errors, they are different things.
Errors are a subclass of Throwable, and indicate a serious problem that the application should not try to deal with.

Tutorial

Exception Types

Exceptions can be checked or unchecked.
Checked exceptions are compile time issues and ones that we can anticipate ahead of time, this allows us to plan for if and when it arrises.
Exception is a parent class of all checked exceptions.
Unchecked exceptions are runtime issues typically caused through bad programming.
For example; trying to address an array at an index that is larger than the arrays size.
If we are throwing a RuntimeException (or any subclass of it) in a method it is not required to specify them in the signatures throw clause.
Below you can find a table providing examples of checked exceptions, unchecked exceptions and errors.

Checked Exception	Unchecked Exception	Errors
FileNotFoundException	ArrayIndexOutOfBoundsException	VirtualMachineErrc
SocketException	ClassCastException	ThreadDeath
UnsupportedDataTypeException	NullPointerException	OutOfMemoryErro
ClassNotFoundException	NumberFormatException	CoderMalfunctionE
NoSuchMethodException	IllegalArgumentException	
NoSuchFieldException		

Handling Exceptions

When handling exceptions there are five keywords that we will use a lot:

- **try** : where we place code that might throw an exception.
- **catch** : used to handle exceptions thrown by code in the try block.
- **finally** : executes the code within its block regardless of whether the exception is handled or not.
- **throw** : used to throw the exception in question.
- **throws** : declares the exception in question.

Try Catch Finally

The try-catch statement is the most common method of handling exceptions and is declared in the following way:

```
public class Exceptions {

    public static void main(String[] args) {
        try {
            /** The system will try to run whatever code is
            * inside here
            */
        } catch(ExceptionType name) {
            /**
            * If an exception occurs in the try block that matches
            * ExceptionType of the catch statement then this block
            * of code will run, name is the reference variable for
            * the exception
            */
        } finally {
            /**
            * Code within this block will always run regardless
            * of whether or not an exception was thrown
            */
        }
    }
}
```

You can use multiple catch statements to handle specific exceptions, but be careful when doing this because the compiler will throw an exception for unreachable code if you try to catch an exception that has already been caught by a previous catch.

```
public class Exceptions {

    public String readFirstLine(String path) throws IOException {
        try {
            BufferedReader br = new BufferedReader(new FileReader(path));
            return br.readLine();
        } catch (IOException e) {
            System.out.println("IOException, message is: " + e.getMessage());
        }
        return "";
    }
}
```

In the above example the method readFirstLine is taking in a String parameter, that will be the path of the file we want to read.
We are also declaring that the method could throw an IOException.
In the try block we are using a buffered reader to open up the file for us, then returning the first line of that file.
BufferedReader is a class from java.io that can be imported to read from files, if you want to learn more about BufferedReader there is a separate module for it.
If the try block throws an IOException then the catch block will catch the exception and any code within the catch block will be run, in this case we are printing the exception message.
Finally the return statement at the bottom of the method is returning an empty string for us, but this line of code is only executed if the try block throws an exception.
If any exception that is not an IOException is thrown then it will not be handled by our code, if we want to handle other exceptions we would have to add more catch statements.

Try With Resources

A try with resources statement is a statement that declares one or more resources, which are objects that must be closed after the program is finished with it.
The resource(s) we declare are put within parentheses after the try statement.

```
public class Exceptions {

    public String readFirstLine(String path) throws IOException {
        try(
            BufferedReader br = new BufferedReader(new FileReader(path))
        ) {
            return br.readLine();
        }
    }
}
```

In the above example we are declaring the resources that we need opened in the parentheses after the try statement.
This eliminates the need of the finally block to close any resources as anything declared within a try-with-resources statement is closed after the try block is run, regardless of whether the try statement completes normally or abruptly.

Exceptions and Inheritance

An overriding method can throw any unchecked exceptions, regardless of whether the overridden method throws exceptions or not.
However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method.
The overriding method can throw checked exceptions that are either declared by the method being overridden or those with less scope than the exceptions declared.
If a constructor is throwing an exception then any subclasses of that class must also throw that same exception.

Custom Exceptions

Java exceptions cover almost all general exceptions that are bound to happen in programming, however sometimes we may require custom exceptions.
Generally we need a custom exception for one of two reasons:

- Business logic exceptions - sometimes we will require custom exceptions that are specific to the business logic and workflow, these help the application users or developers understand what the problem is.
- To catch and provide specific treatment to a subset of existing Java exceptions.

To create a custom exception we will need to create a new class with the name we want our exception to be and have it inherit from the Exception class.
We can do this with the extends keyword, if you want to learn more about inheritance we have a separate module, but it essentially means that the class members from the Exception class will be passed down to our custom exception.
If we are reading from a file and a FileNotFoundException is being thrown, it is not clear what the exact reason behind the exception is, it could be that the file does not exist or the incorrect file name was given.
If we want to have a more specific exception like IncorrectPathException then the new class for our exception will look something like the following:

```
public class IncorrectPathException extends Exception {

    public IncorrectPathException(String errorMessage) {
        super(errorMessage);
    }
}
```

The above example is a new class for our exception called IncorrectPathException, which is inheriting from Exception and has a constructor that takes in a String and passes that String to the super classes constructor.
The constructor is used to instantiate the object, if you want to learn more about constructors there is a separate module for it.
This is all we need to do to define a custom exception, and so now we can use this exception within our code.

```
public class Exceptions {

    public String readFirstLine(String path) throws FileNotFoundException {
        try {
            BufferedReader br = new BufferedReader(new FileReader(path));
            return br.readLine();
        } catch (FileNotFoundException e) {
            if(!isCorrectPath(path)) {
                throw new IncorrectPathException("Incorrect Path : " + path);
            }
        }
        return "";
    }

    public boolean isCorrectPath(String path) {
        // Some code to check path is correct
    }
}
```

The above example is using a try-with-resources statement using a BufferedReader as a resource, and then trying to read the first line in the file that has been passed through (the file is passed as an entire path).

If the try block fails and a FileNotFoundException is thrown then we catch the exception and go into the catch block code.
Within the catch block we have a conditional that is calling the method isCorrectPath, this method will have some sort of check to see if the path provided was valid, and if it was not then we will instead throw our IncorrectPathException.
However within this example we are losing the root cause of the exception as we are passing through a completely custom String, so what we can do is overload the IncorrectPathException class so that it has multiple constructors, one of which will be the constructor we already have, but we can also add a constructor that takes in a Throwable as a parameter.
Overloading allows us to have multiple constructors for a class so long as they have either different parameter types or a different amount of variables, if you want to learn more about overloading we have a separate module for polymorphism.
Taking a Throwable in as a parameter will allow us to pass the original exception through to our custom exception so that we can still access things like the stack trace.
So now our IncorrectPathException class will look something like this:

```
public class IncorrectPathException extends Exception {

    public IncorrectPathException(String errorMessage) {
        super(errorMessage);
    }

    public IncorrectPathException(String errorMessage, Throwable err) {
        super(errorMessage, err);
    }
}
```

We can also add a default constructor if we wanted so that we don't need to pass any parameters whenever we want to throw the exception.
We can also create custom unchecked exceptions if we wanted to, it would look similar to the above example, only instead of extending Exception, we would extend RuntimeException.

Exercises

Division With Exceptions

1. Create a program that can take two ints as input from the user and can produce the division of these numbers.
   You can use the Maths project created earlier.
   This method will need to catch two specific exceptions as well as an overall third exception.

2. Handle each of these exceptions within the method and handle them in different ways.

3. Recreate the division method to throw your own bespoke exception.
   This exception will be thrown if the user tries to divide a number (a) by a larger number (b).
   - Create a separate class that will be your exception, this class will extend the Exception class.
   - Create and implement the division method so that it takes account of your new exception.
   - Handle this exception in varying ways to show the flexibility of exception handling.