TASK

Defensive Programming

Visit our website

Introduction

WELCOME TO THE DEFENSIVE PROGRAMMING TASK!

Debugging is essential to any Software Developer! In this task, you will learn about some of the most commonly encountered errors.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

WHAT IS DEFENSIVE PROGRAMMING?

Defensive programming is an approach to writing code where the programmer tries to anticipate problems that could affect the program and then takes steps to defend the program against these problems. MANY problems could cause a program to run unexpectedly!

Some of the more common types of problems to look out for are listed below:

- **User errors:** The people that use your application will act unexpectedly! They will do things like entering string values where you expect numbers. Or they may enter numbers that cause calculations in code to crash (e.g. the prevalent divide by 0 error). Otherwise, they may press buttons at the wrong time or more times than you expect. As a developer, anticipate these problems and write code that can handle these situations. For example, check all user input.

- **Errors caused by the environment:** Write code that will handle errors in the development and production environment. For example, in the production environment, your program may get data from a database that is on a different server. Code should be written in such a way that it deals with the fact that some servers may be down, the load of people accessing the program is higher than expected, etc.

- **Logical errors with the code:** Besides external problems that could affect a program, a program may also be affected by errors within the code. All code should be thoroughly tested and debugged. You will learn to write automated tests in the next level of this Bootcamp.

IF STATEMENTS FOR VALIDATION

Many of the programming constructs that you have already learned can be used to write defensive code. For example, if you assume that any user of your system will be younger than 150 years old, you could use a simple *if statement* to check that someone enters a valid age. Exception handling is also essential for defensive programming.

EXCEPTION HANDLING

Oracle defines an exception as "an event that occurs during the execution of a program that disrupts the normal flow of instructions". When an exception occurs, a special object that describes the exception is created.

There are two main types of exceptions: **unchecked** and **checked** exceptions (Oracle, 2019):

- Unchecked exceptions can be either errors or runtime exceptions.
    - Errors are problems that occur outside of the program. A Programmer usually cannot anticipate or fix errors. An example of an error would be when a program fails to read an existing file because of a system or hardware malfunction (Oracle, 2019).
    - Logical errors within the code often cause runtime exceptions. Passing a null value where an object is required is an example of a runtime error. Runtime exceptions should be fixed in the code whenever possible. Since runtime exceptions are caused by situations that depend on the state of your program, they cannot be picked up beforehand by Java, and so it is up to you to avoid them. An example of a runtime error:

```
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at Example.main(Example.java:7)
```

- Checked exceptions are exceptions that a programmer should be able to anticipate and handle. Checked exceptions are handled using *try-catch* blocks.

Consider the code example of handling a checked exception below. You are already familiar with most of the code. In the example, we try to read a text file called **filename.txt** in the same folder as the Java file being executed. If the text file cannot be opened, a runtime error will occur. If the file is not found, a FileNotFoundException will be thrown.

```
import java.io.File;  // Import the File class
import java.io.FileNotFoundException;  // Import this class to handle errors
import java.util.Scanner; // Import the Scanner class to read text files
```

```
public class exceptionHandlingEg {
   static File myObj;
   static Scanner myReader;

   public static void main(String[] args) {
       try {

         myObj = new File("filename.txt");
           myReader = new Scanner(myObj);

           while (myReader.hasNextLine()) {
             String data = myReader.nextLine();
             System.out.println(data);
           }
       }
       catch (FileNotFoundException e) {
         System.out.println("An error occurred. The file could not be
found!");
         System.out.println();
         System.out.println(e.getMessage());
         System.out.println();
         e.printStackTrace();
       }
       finally {
         myReader.close();
       }
   }
}
```

In the code above, defensive programming is used. The coder has anticipated that the file may not be found. Therefore, the code handles the exception.

To handle exceptions:

1. Create a **try block** that contains the code that may cause a runtime error to occur. Code in the *try* block is executed as normal. If an exception occurs during the execution of the code in the *try* block, the *catch* block is called.

2. Create a **catch block** that handles the exception. Therefore, "each catch block is an exception handler that handles the type of exception indicated by its argument" (Oracle, 2019). Code in the *catch* block is only executed if an exception is thrown. If no exception occurs when the code in the *try* block is executed, the *catch* block will be ignored. You must decide how to handle exceptions. You may choose to display error messages to the user, to log error messages somewhere, to alert an administrator about the error, etc. In

the example above, an error message is displayed. It is possible to have more than one *catch* block associated with a *try* block. As you can see in the code example, an exception object is passed as an argument to the *catch* block. The exception object is discussed in more detail in the next subheading.

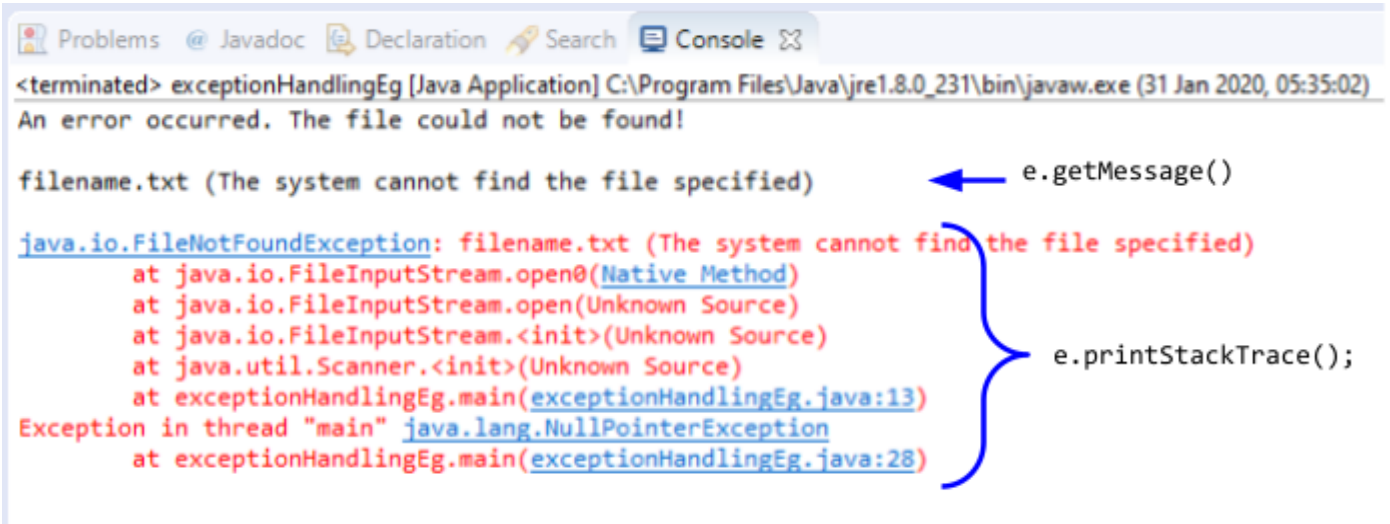
3. The **finally block** is optional. Code in the *finally* block will always be executed, whether an exception occurred or not. The *finally* block is a good place to put code that must be executed even if an error occurs. In the example below, whether the file is successfully read or not, the Scanner object is closed in the *finally* block.

Beware! Do not be tempted to overuse *try-catch* blocks! If you can anticipate and fix potential problems without using *try-catch* blocks, do so. For example, don't use *try-catch* blocks to avoid writing code that properly validates user input.

EXCEPTION OBJECTS

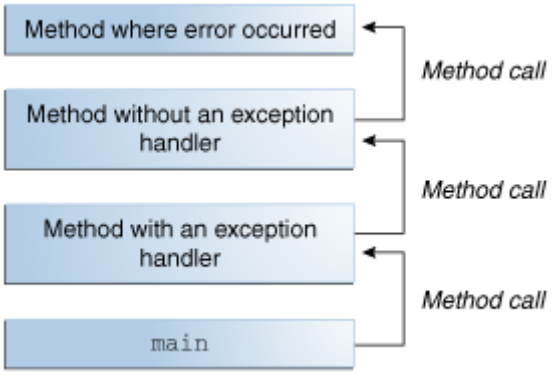
When an exception occurs, an exception object is created. The exception object contains information about the error.

Consider the code example in the previous subheading again. Notice that an object called `e` of type `FileNotFoundException` is a parameter of the *catch* block. The exception object contains information about the error. For example, `e.getMessage()` returns the error message associated with the exception. See an example of the error message returned in the image below:



Notice that in the example, `e.printStackTrace();` is also used. The Stack being referred to is a call stack. The call stack is the ordered list of methods called to get

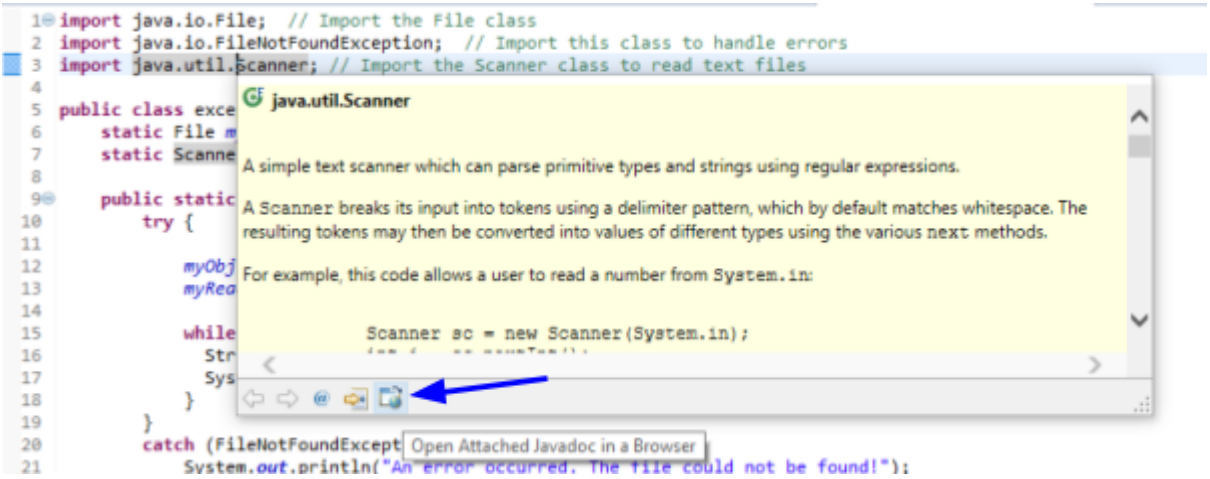
to the method where the error occurred. Examining the call stack can help identify the cause of the error.



The Call Stack (Oracle, 2019a)

How do you know what exception object to use?

One of the easiest ways to see what exceptions to anticipate and handle in your code is by examining the Javadoc for the classes that you import for your program. To access the Javadoc for a class in Eclipse, hover over the name of the class in the import statement in your code. Press F2 to focus on the window that appears (see the image below).



Select the button that allows you to “Open Attached Javadoc in a Browser”. Scroll down in this Javadoc to see which exceptions may be thrown by the class. As shown in the image below, the Javadoc also describes the exception object.

Scanner

```
public Scanner(Path source,
               String charsetName)
        throws IOException
```

Constructs a new Scanner that produces values scanned from the specified file. Bytes from the file are converted into characters using the specified charset.

Parameters:

`source` - the path to the file to be scanned

`charsetName` - The encoding type used to convert bytes from the file into characters to be scanned

Throws:

`IOException` - if an I/O error occurs opening source

`IllegalArgumentException` - if the specified encoding is not found

Compulsory Task

Follow these steps:

- Open the Java file **Errors.java**.
- Attempt to compile the program. You'll notice the compilation will fail due to some errors. Fix all the compilation errors and compile the program again. Every time you fix an error, add a comment to the line you fixed and indicate which of the three types of errors it was.
- Now run the program. Once again, you'll encounter some errors (this time, runtime errors). Fix these errors and run the program once again to see if it works.
- Now run the program and notice the output. You'll notice that there are some logical errors in the code. Fix these errors and run the program once again to see if it now correctly displays what it should.

Rate us

Share your thoughts

HyperionDev strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think that the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.

References:

Loudoun County Public Schools. (n.d.). *Stepping through the main method*. Retrieved from https://www.lcps.org/cms/lib/VA01000195/Centricity/Domain/5120/Debugger.docx

Oracle. (2019). *The Catch or Specify Requirement*. Retrieved from The Java Tutorials: https://docs.oracle.com/javase/tutorial/essential/exceptions/catchOrDeclare.html

Oracle. (2019a). What Is an Exception? Retrieved from The Java Tutorials: https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html