TASK

Towards defensive programming II

Visit our website

Introduction

WELCOME TO THE DEFENSIVE PROGRAMMING TASK!

Debugging is an essential skill for any Software Developer! In this task, you will learn about some of the most commonly encountered errors. You will also learn to use your IDE to debug your code more effectively.

Get in touch

Connect for support

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

The best way to get help is to login to Discord at **https://discord.com/invite/hyperdev** where our specialist team is ready to support you.

Our team 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 in an unexpected way!

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 because they are human beings! They will do things like entering string values where you expect numbers, or entering numbers that cause calculations in code to crash (e.g. the prevalent 'divide by 0' error). They may also press buttons at the wrong time or more times than you expect. As a developer, you need to anticipate these problems and write code that can handle these situations, for example by checking 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 may be 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.

TRY AND CATCH BLOCK

When handling errors there are 2 main blocks of code that are used to ensure that certain actions are taken when the error occurs, namely the *try block* and the *catch block*. The try block is the part of the code that we are trying to execute if no errors occur. The catch block, therefore, is the code that we specify needs to be executed if an error does occur in the try block.

The JavaScript statements try and catch come in pairs:

try {
* Block of code to try*
}
catch(err) {
* Block of code to handle errors*
}

Let's look at the following example:

```
while (true) {
    try {
        let x = prompt("enter a number");
        if (x == "") {
            throw "You have not entered anything";
        }
        if (isNaN(x)) { // check whether input is not a number
            throw x + " is not a number";
        }
        x = Number(x);
        console.log(x);
        break;
    } catch (err) {
        console.log(err);
    }
}
```

In the above code example, we have a while loop used to take user input. The try block gets executed first. If the input is a valid number, the numeric value will be assigned to the **x** variable, and then the loop will break.

However, if the user input is a string that is not a valid number, an error message will be thrown. The catch block will then be executed, thereby printing the message to the terminal. It is worth noting that any code can be executed in the catch block and it does not have to be a print statement. After the message is printed to the terminal, the loop will start from the try block again until the user enters a valid number and the loop can break.

TRY - CATCH - FINALLY

Check the example code below:

```
try{
```

```
    //The try block. You can have more than 1 throw in this block.
    let numb1 = prompt("First number");
    if(isNaN(numb1)){
        throw numb1 + " is not a number";
    }
    let numb2 = prompt("Second Number");
    if (isNaN(numb2)){
        throw numb2 + " is not a number";
    }
    if (numb2 == 0){
        throw "Can't divide by 0";
    }
    numb1 = Number(numb1);
    numb2 = Number(numb2);
    console.log(numb1+ " / " + numb2 + " = "+ (numb1/numb2));
}catch(err){
    //This block will be executed if and when the try block throws an error
    console.log(err);
}
finally{
    //This block will be executed whether the catch block was executed or not.
    console.log("The program will continue from here. ");
}
```

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. 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, log error messages somewhere, alert an administrator about the error, etc. In the example above, an error message is logged to the console. As you can see in the code example, an exception object is passed as an argument to the *catch* block.

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 above the program

will log the message *"The program will continue from here. "*, regardless of whether the error occurred or not.

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.

Compulsory Task 1

Create a file called **defensive.js**.

In this file, you are going to write a program that will be used to calculate distance, time, or speed.

- First ask a user to choose whether to calculate distance, time, or speed.
- If the user chooses distance:
  - prompt the user for time
  - prompt the user for speed
  - calculate a distance using the formula: *distance = speed × time*.
- If the user chooses the time:
  - prompt the user for distance
  - prompt the user for speed
  - calculate a time using the formula: *time = distance ÷ speed*.
- If the user chooses the speed:
  - prompt the user for distance
  - prompt the user for time
  - calculate a speed using the formula: *speed = distance ÷ time*.

Use defensive programming when writing this program to anticipate errors and handle them. As always remember to comment your code making it clear what you are doing.

If you are having any difficulties, please feel free to contact our specialist team **on Discord** for support.

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.