17.1. Introduction

Errors are a part of coding. Occasionally, we make mistakes as programmers. However, we are always trying to fix those mistakes by reading different resources, examining a list of error messages (also called the **stacktrace**), or asking for help.

Earlier in this course, we learned about two different types of errors: runtime and logic. A logic error is when your program executes without breaking, but doesn't behave the way you thought it would. These logic errors usually require you to consider how you are going about solving the issue to resolve. Runtime errors are when your program does not run correctly, and an exception is raised.

An **exception** is a runtime error in which a name and message are displayed to provide more information about the error.

17.1.1. Exceptions and Errors

In JavaScript a runtime error and an exception are the same thing and can be used interchangeably. This can cause confusion because a logic error is not an exception!

17.1.2. Error Object

When a runtime error, also known as an exception, is raised JavaScript returns an `Error` object. An Error Object has two properties: a name and a message. The name refers to the type of error that occurred, while the message gives the user information on why that exception occurred.

JavaScript has built-in exceptions with pre-defined names and messages, however, JavaScript also gives you the ability to create your own error messages.

You have undoubtedly experienced various Exceptions already throughout this class. Let's look at a few common Exceptions.

17.1.3. Common Exceptions

JavaScript has some built-in Exceptions you may have already encountered in this class.

One of the most common errors in Javascript is a `SyntaxError` which is thrown when we include a symbol JavaScript is not expecting.

Example

```
console.log("This is" an example);
```

Console Output

```
SyntaxError: missing ) after argument list
```

We put our second quotation mark in the incorrect place. JavaScript does not know what to do with the second half of our phrase and throws a `SyntaxError` with the message: `missing ) after argument list`.

A `ReferenceError` is thrown when we try to use a variable that has not yet been defined.

Example

```
console.log(x[0]);
```

Console Output

```
ReferenceError: x is not defined
```

We attempt to print out the first element in the variable x, but we never declared x. JavaScript throws a `ReferenceError` with the message: `x is not defined`.

A `TypeError` is thrown when JavaScript expects something to be one type, but the provided value is a different type.

Example

```
const a = "Launch";

a = "Code";
```

Console Output

```
TypeError: invalid assignment to const 'a'
```

In this case, we declare a constant as the string "Launch", and then try to change the immutable variable to "Code". JavaScript throws a `TypeError` with the message: `invalid assignment to const 'b'`.

Exceptions give us a way to provide more information on how something went wrong. JavaScript's built-in Exceptions are regularly used in the debugging process.

There are more built-in Exceptions in Java, you can read more by referencing the MDN Errors Documentation or W3Schools JavaScript Error (scroll down to the Error Object section).

In the next section we will learn how to raise our own exceptions using the `throw` statement.

17.1.4. Check Your Understanding

Question

What is the difference between a runtime error, and a logic error?

Question

What are some of the common errors included in JavaScript?

17.2. Throw

In most programming languages, when the compiler or interpreter encounters code it doesn't know how to handle, it **throws** an exception. This is how the compiler notifies the programmer that something has gone wrong. Throwing an exception is also known as *raising* an exception.

JavaScript gives us the ability to raise exceptions using the `throw` statement. One reason to throw an exception is if your code is being used in an unexpected way.

17.2.1. Throw Default Error

We can throw a default Error by using the `throw` statement and passing in a string description as a argument.

Example

```
1 throw Error("You cannot divide by zero!");
```

repl.it

Console Output

```
Error: You cannot divide by zero!
at evalmachine.<anonymous>:1:7
at Script.runInContext (vm.js:133:20)
at Object.runInContext (vm.js:311:6)
at evaluate (/run_dir/repl.js:133:14)
```

The error text displays the error name, and it contains details about where the error was thrown. The text `at evalmachine.<anonymous>:1:7` indicates that the error as thrown from line 1, which we know is true because our example only has one line of code.

17.2.2. Pre-existing Error

JavaScript also gives us the power to throw a more specific type of error.

Example

```
throw SyntaxError("That is the incorrect syntax");
```

Console Output

```
SyntaxError: That is the incorrect syntax
```

JavaScript gives us flexibility by allowing us to raise standard library errors and to define our own errors. We can use exceptions to allow our program to break and provide useful information as to why something went wrong.

17.2.3. Custom Error

JavaScript will also let you define new types of Errors. You may find this helpful in the future, however, that goes beyond the scope of this class.

17.2.4. Check Your Understanding

Question

What statement do we use to raise an exception?

Question

How do we customize the message of an exception?

17.3. Exceptions as Control Flow

Runtime errors occur as the program runs, and they are also called exceptions. Exceptions are caused by referencing undeclared variables and invalid or unexpected data.

17.3.1. Control Flow

The **control flow** of a program is the order in which the statements are executed. Normal control flow runs from top to bottom of a file. An exception breaks the normal flow and stops the program. A stopped program can no longer interact with the user. Luckily JavaScript provides a way to anticipate and handle exceptions.

17.3.2. Catching an Exception

JavaScript provides `try` and `catch` statements that allow us to keep our programs running even if there is an exception. We can tell JavaScript to *try* to run a block of code, and if an exception is thrown, to *catch* the exception and run a specific block of code. Anticipating and catching the exception makes the exception now part of the control flow.

Note

Catching an exception is also known as *handling* an exception.

Example

In this example there is an array of animals. The user is asked to enter the index for the animal they want to see. If the user enters an index that does NOT contain an animal, the code will throw an `TypeError` when `name` is referenced on an undefined value.

There is a `try` block around the code that will throw the `TypeError`. There is a `catch` block that catches the error and contains code to inform the user that they entered an invalid index.

```
const input = require('readline-sync');

let animals = [{name: 'cat'}, {name: 'dog'}];
let index = Number(input.question("Enter index of animal:"));

try {
  console.log('animal at index:', animals[index].name);
} catch(TypeError) {
  console.log("We caught a TypeError, but our program continues to run!");
  console.log("You tried to access an animal at index:", index);
}

console.log("the code goes on...");

```

repl.it

Console Output

If the user enters `9`:

```
Enter index of animal: 9
We caught a TypeError, but our program continues to run!
You tried to access an animal at index: 9
the code goes on...
```

If the user enters `0`:

```
Enter index of animal: 0
animal at index: cat
the code goes on...
```

Tip

`catch` blocks only execute if an exception is thrown

17.3.3. Finally

JavaScript also provides a `finally` block which can be used with `try` and `catch` blocks. A `finally` block code runs after the `try` and `catch`. What is special about `finally` is that `finally` code block ALWAYS runs, even if an exception is NOT thrown.

Example

Let's update the above example to print out the index the user entered. We want this message to be printed EVERY time the code runs. Notice the `console.log` statement on line 11.

```
 const input = require('readline-sync');
1
2let animals = [{name: 'cat'}, {name: 'dog'}];
3let index = Number(input.question("Enter index of animal:"));
4
5try {
6   console.log('animal at index:', animals[index].name);
7} catch(TypeError) {
8   console.log("We caught a TypeError, but our program continues to run!");
9} finally {
10   console.log("You tried to access an animal at index:", index);
11}
12
13console.log("the code goes on...");
14
```

repl.it

Console Output

If the user enters `7`:

```
Enter index of animal: 7
We caught a TypeError, but our program continues to run!
You tried to access an animal at index: 7
the code goes on...
```

If the user enters `1`:

```
Enter index of animal: 1
animal at index: dog
You tried to access an animal at index: 1
the code goes on...
```

17.3.4. Check Your Understanding

Question

What statement do we use if we want to attempt to run code, but think an exception might be thrown?

a. `catch`
b. `try`
c. `throw`
d. `finally`

Question

How do you handle an exception that is thrown?

a. With code placed within the `try` block.
b. With code placed within the `catch` block.
c. With code placed within a `throw` statement.
d. With code placed within the `finally` block.

Question

What statement do you use to ensure a code block is executed regardless if an exception was thrown?

a. `throw`
b. `catch`
c. `try`
d. `finally`

17.4. Exercises

17.4.1. Zero Division: Throw

Write a function called divide that takes two parameters: a `numerator` and a `denominator`.

Your function should return the result of `numerator / denominator`.

However, if the `denominator` is zero you should throw an error informing the user they attempted to divide by zero. The error text should be `Attempted to divide by zero.`

Note

Hint: You can use an `if / throw` statement to complete this exercise.

17.4.2. Test Student Labs

A teacher has created a `gradeLabs` function that verifies if student programming labs work. This function loops over an array of JavaScript objects that *should* contain a `student` property and `runLab` property.

The `runLab` property is expected to be a function containing the student's code. The `runLab` function is called and the result is compared to the expected result. If the result and expected result don't match, then the lab is considered a failure.

```
function gradeLabs(labs) {
  for (let i=0; i < labs.length; i++) {
     let lab = labs[i];
     let result = lab.runLab(3);
     console.log(`${lab.student} code worked: ${result === 27}`);
  }
}

let studentLabs = [
  {
     student: 'Carly',
     runLab: function (num) {
        return Math.pow(num, num);
     }
  },
  {
     student: 'Erica',
     runLab: function (num) {
        return num * num;
     }
  }
];

gradeLabs(studentLabs);

```

The `gradeLabs` function works for the majority of cases. However what happens if a student named their function incorrectly? Run `gradeLabs` and pass it `studentLabs2` as defined below.

```
let studentLabs2 = [
```

```
  {
     student: 'Jim',
     myCode: function (num) {
        return num + num;
     }
  },
  {
     student: 'Jessica',
     runLab: function (num) {
        return Math.pow(num, num);
     }
  }
];

gradeLabs(studentLabs2);
```

Upon running the second example, the teacher gets a `TypeError: lab.runLab is not a function`.

Your task is to add a `try catch` block inside of `gradeLabs` to catch an exception if the `runLab` property is not defined.

17.5. Studio: Strategic Debugging

17.5.1. Summary

At this point, we have seen a lot of different types of errors. We have possibly created logic errors or syntax errors and now, we have just learned about the `Error` object in JavaScript. The goal of this studio is for us to develop strategies for debugging so that we can get rid of the bugs and get back to coding!

17.5.2. Activity

Think of a bug you have seen in your code. This could be the time you dropped a keyword when initializing a variable or misused a method.

1. Take some time to discuss with the group what your error was and how you solved it. Did you talk to a TA to get it? Did you find a great resource online that was helpful?

Your TA will go over the pros and cons of different resources that can help you resolve the error. You will then go over a general strategy to start debugging your errors.

17.5.3. Debugging Process

Your TA will go over this process with you and how it could help you debug more strategically. This process reflects what we have found works best for us and many students, however, as you grow as a programmer, you may find something works better for you. That is fine! Every programmer has their own process for debugging based off of their experiences and how their mind works.

1. Check the *stacktrace* to read the error message and see where it occurred.
2. If you see the error, fix it on that line and recompile.
3. If you cannot see the error, Google the error message.
4. Check any relevant StackOverflow posts in the results.
5. If the error is related to built-in methods or objects, also search for those in the official documentation.
6. If the error is related to something that cannot be done in that particular language, look at the responses to each comment before trying to replicate proposed solutions. Solutions can oftentimes go out of date and responses will tell you if that is the case or simply if it is a bad solution.