Week 05 Notes:

TESTING AND DEBUGGING:

ERRORS, EXCEPTIONS AND WARNINGS:

* System error ― there’s a problem with the system or external devices with which the program is interacting.
* Programmer error ― the program contains incorrect syntax or faulty logic; it could even be as simple as a typo.
* User error ― the user has entered data incorrectly, which the program is unable to handle.

As programmers, we often have little influence over how external systems work, so it can be difficult to fix the root cause of system errors. Despite this, we should still be aware of them and attempt to reduce their impact by working around any problems they cause. Programmer errors are our responsibility, so we must ensure they are minimized as much as possible and fixed promptly. We also should try to limit user errors by predicting any possible interactions that may throw an error, and ensure they are dealt with in a way that doesn’t negatively affect the user experience. It might even be argued that user errors are in fact also programmer errors, because the program should be designed in a way that prevents the user from making the error.

EXCEPTIONS:

An exception is an error that produces a return value that can then be used by the program to deal with the error. For example, trying to call a method that is nonexistent will result in a reference error that raises an exception, as you can see in the example below when we try to call the mythical unicorn() function:

Texto

Descripción generada automáticamente con confianza media

STACK TRACES:

An exception will also produce a **stack trace**. This is a sequence of functions or method calls that lead to the point where the error occurred. It’s often not just a single function or method call that causes an error. A stack trace will work backwards from the point at which the error occurred to identify the original function or method that started the sequence. The example below shows how a stack trace can help you find where an error originates from:

Texto

Descripción generada automáticamente

WARNINGS:

A warning can occur if there’s an error in the code that isn't enough to cause the program to crash. This means the program will continue to run after a warning. This might sound good, but it can be problematic, since the issue that produced the warning may cause the program to continue running incorrectly.

An example of a mistake that could cause a warning is assigning a value to a variable that’s undeclared:

Texto

Descripción generada automáticamente

When a runtime error occurs in the browser, the HTML will still appear, but the JavaScript code will stop working in the background, which isn’t always obvious at first. If a warning occurs, the JavaScript will continue to run (although possibly incorrectly).

THE IMPORTANCE OF TESTING AND DEBUGGING:

or this reason, a ninja programmer should ensure that the code they write fails loudly in development so any errors can be identified and fixed quickly. In production, a ninja programmer should try to make the code fail gracefully (although not completely silently ― we still need to know there’s an error), so the user experience is not affected, if possible. This is achieved by making sure exceptions are caught and dealt with, and code is tested rigorously.

STRICT MODE:

ECMAScript 5 introduced a strict mode that produces more exceptions and warnings and prohibits the use of some deprecated features. This is due to the fact that strict mode considers coding practices that were previously accepted as just being 'poor style' as actual errors.

Increasing the chance of errors might seem like a bad idea at first, but it’s much better to spot errors early on, rather than have them cause problems later. Writing code in strict mode can also help improve its clarity and speed, since it follows conventions and will throw exceptions if any sloppy code practices are used.

Not using strict mode is often referred to as 'sloppy mode' as it’s forgiving of sloppy programming practices. Strict mode encourages a better quality of JavaScript to be written that befits a ninja programmer, so its use is recommended.

Strict mode simply requires the following string to be added to the first line of a

'use strict';

This will be picked up by any JavaScript engine that uses strict mode. If the engine does not support strict mode, this string will simply be ignored.

To see it in action, if you try to assign a value to a variable that is undeclared in strict mode, you'll get an exception, instead of a warning:

Texto

Descripción generada automáticamente

You can even use strict mode on a per-function basis by adding the line inside a function. Strict mode will then only be applied to anything inside that function:

Texto

Descripción generada automáticamente

In fact, the recommended way to invoke strict mode is to place all your code into a self-invoking function (covered in more detail in Chapter 12), like so:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Placing 'use strict' at the beginning of a file will enforce strict mode on all the JavaScript in the file. And if you’re using anybody else’s code, there’s no guarantee they’ve coded in strict mode. This technique will ensure that only your code is forced to use strict mode.

ES6 introduced JavaScript modules (covered later in chapter 15). These are self-contained pieces of code that are in strict mode by default, so the 'use strict' declaration is not required.

LINTING TOOLS:

Linting tools such as [JS Lint,](http://jslint.com/)[JS Hint,](http://jshint.com/) and [ES Lint](http://eslint.org/) can be used to test the quality of JavaScript code, beyond simply using strict mode. They are designed to highlight any sloppy programming practices or syntax errors, and will complain if certain style conventions are not followed, such as how code is indented. They can be very unforgiving and use some opinionated coding conventions, such as not using the ++ and -- increment operators (in the case of JS Lint). Linting tools are also useful for enforcing a programming **style guide**. This is particularly useful when you are working in a team, as it ensures everybody follows the same conventions.

It's possible to add a linting tool as a text-editor plugin; this will then highlight any sloppy code as you type. Another option is to use an online linting tool that allows you to simply paste onto a page for feedback. Another option is to install linting software on your computer using npm. This can then be run as part of your workflow.

Passing a lint test is no guarantee that your code is correct, but it will mean it will be more consistent and less likely to have problems.

FEATURE DETECTION:

The recommended way to determine browser support for a feature is to use feature detection. This is done using an if statement to check whether an object or method exists before trying to actually call the method. For example, say we want to use the shiny new holoDeck API (as far as I know, this doesn't actually exist ... yet), we would wrap any method calls inside the following if block:

Texto

Descripción generada automáticamente

DEBUGGING IN THE BROWSER:

THE TRUSTY ALERT:

Using alerts for debugging was the only option in the past, but JavaScript development has progressed since then and their use is discouraged for debugging purposes today.

USING THE CONSOLE:

We've already seen and used the console.log() method.

* The console.trace() method will log an interactive stack trace in the console. This will show the functions that were called in the lead up to an exception occurring while the code is running.

[This SitePoint post](http://www.sitepoint.com/three-little-known-development-console-api-methods/) also lists a few other useful but little-known methods of the console object.

DEBUGGING TOOLS:

One of the most useful commands is the debugger keyword. This will create a breakpoint in your code that will pause the execution of the code and allow you to see where the program is currently up to. You can also hover over any variables to see what value they hold at that point. The program can then be restarted by clicking on the 'play' button.

The example below shows how the debugger command can be used in the amIOldEnough() function. If you try entering the code below into your browser's console, then invoke the amIOldEnough() function, the browser's debugging tool will automatically kick in and you'll be able see the value of the age variable by hovering over it:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Remember to remove any references to the debugger command before shipping any code, otherwise the program will appear to freeze when people try to use it!

ERROR OBJECTS:

An error object can be created by the host environment when an exception occurs, or it can be created in the code using a constructor function, like so:

QUESTION:

WHAT DOES HOST ENVIRONMENT REFERS TO?

Texto

Descripción generada automáticamente

This constructor function takes a parameter that’s used as the error message:

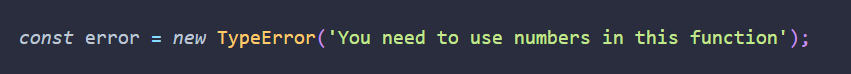
Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

There are seven more error objects used for specific errors:

* EvalError is not used in the current ECMAScript specification and only retained for backwards compatibility. It was used to identify errors when using the global eval() function.
* RangeError is thrown when a number is outside an allowable range of values.
* ReferenceError is thrown when a reference is made to an item that doesn’t exist. For example, calling a function that hasn't been defined.
* SyntaxError is thrown when there’s an error in the code’s syntax.
* TypeError is thrown when there’s an error in the type of value used; for example, a string is used when a number is expected.
* URIError is thrown when there’s a problem encoding or decoding the URI.
* InternalError is a non-standard error that is thrown when an error occurs in the JavaScript engine. A common cause of this too much recursion.

These error objects can also be used as constructors to create custom error objects:



All error objects have a number of properties, but they’re often used inconsistently across browsers. The only properties that are generally safe to use are:

* The name property returns the name of the error constructor function used as a string, such as 'Error' or 'ReferenceError'.
* The message property returns a description of the error and should be provided as an argument to the Error constructor function.
* The stack property will return a stack trace for that error. This is a non-standard property and it’s recommended that it is not safe to use in production sites.

THROWING EXCEPTIONS:

Hasta ahora, hemos visto errores que el motor de JavaScript lanza automáticamente cuando ocurre un error. También es posible lanzar sus propias excepciones usando la  throw declaración. Esto permitirá que cualquier problema en su código se destaque y se resuelva, en lugar de acechar silenciosamente en segundo plano.

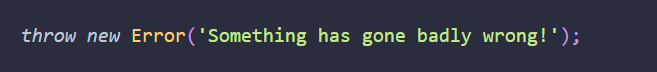
La  throw declaración se puede aplicar a cualquier expresión de JavaScript, lo que hace que se detenga la ejecución del programa. Por ejemplo, todas las siguientes líneas de código harán que un programa se detenga:

*throw* 2;

*throw* 'Up';

*throw* { toys: 'out of pram' };

It is best practice, however, to throw an error object. This can then be caught in a catch block, which is covered later in the chapter:



As an example, let’s write a function called squareRoot() to find the square root of a number. This can be done using the Math.sqrt() method, but it returns NaN for negative arguments. This is not strictly correct (the answer should be an imaginary number, but these are unsupported in JavaScript). Our function will throw an error if the user tries to use a negative argument:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Texto

Descripción generada automáticamente

EXCEPTION HANDLING:

When an exception occurs, the program terminates with an error message. This is ideal in development as it allows us to identify and fix errors. In production, however, it will appear as if the program has crashed, which does not reflect well on a ninja programmer.

It is possible to handle exceptions gracefully by catching the error. Any errors can be hidden from users, but still identified. We can then deal with the error appropriately ― perhaps even ignore it ― and keep the program running.

TRY, CATCH, AND FINALLY:

If we suspect a piece of code will result in an exception, we can wrap it in a try block. This will run the code inside the block as normal, but if an exception occurs it will pass the error object that is thrown onto a catch block. Here’s a simple example using our squareRoot() function from earlier:

Pantalla negra con letras blancas

Descripción generada automáticamente

The code inside the catch block will only run if an exception is thro wn inside the try block. The error object is automatically passed as a parameter to the catch block. This allows us to query the error name, message and stack properties, and deal with it appropriately. In this case, we actually return a string representation of an imaginary number:

Texto

Descripción generada automáticamente

A finally block can be added after a catch block. This will always be executed after the try or catch block, regardless of whether an exception occurred or not. It is useful if you want some code to run in both cases. We can use this to modify the imaginarySquareRoot() function so that it adds "+ or -" to the answer before returning it:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

TESTS:

Testing is an important part of programming that can often be overlooked. Writing good tests means your code will be less brittle as it develops, and any errors will be identified early on.

A test can simply be a function that tests a piece of code runs as it should. For example, we could test that the squareRoot() function that we wrote earlier returns the correct answer with the following function:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

TEST-DRIVEN DEVELOPMENT:

**Test-driven development** (TDD) is the process of writing tests before any actual code. Obviously these tests will initially fail, because there is no code to test. The next step is to write some code to make the tests pass. After this, the code is refactored to make it faster, more readable, and remove any repetition. The code is continually tested at each stage to make sure it continues to work. This process should be followed in small piecemeal chunks every time a new feature is implemented, resulting in the following workflow:

1. Write tests (that initially fail)
2. Write code to pass the tests
3. Refactor the code
4. Test refactored code
5. Write more tests for new features

TESTING FRAMEWORKS:

Testing frameworks provide a structure to write meaningful tests and then run them. There are a large number of frameworks available for JavaScript, but we’ll be focusing on the Jest framework.

JEST:

[Jest](https://facebook.github.io/jest/) is a TDD framework, created by Facebook, that has gained a lot of popularity recently. It makes it easy to create and run tests by providing helper methods for common test assertions.

To use Jest, first we need to install it using npm. Enter the following command in a terminal:

Texto

Descripción generada automáticamente

This should install Jest globally. To check everything worked okay, try running the following command to check the version number that has been installed:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Next we'll create an example test to see if it works. Let's write a test to see if our squareRoot() function from earlier works. Create a file called squareRoot.test.js and add the following code:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

This file contains the squareRoot() function that we are testing, as well as a test() function. The first parameter of the test() function is a string that describes what we are testing, in this case that ‘square root of 4 is 2’. The second parameter is an anonymous function that contains a function called expect(), which takes the function we’re testing as an argument, and returns an **expectation object**. The expectation object has a number of methods called **matchers**. In the example above, the matcher is toBe(), which tests to see if the value returned by our squareRoot() function is the same as the value provided as an argument (2, in this case). These matchers are named so they read like an English sentence, making them easier to understand (even for non-programmers), and the feedback they provide more meaningful. The example above almost reads as ‘expect the square root of 4 to be 2’. It’s important to recognize that these are just functions at the end of the day, so they behave in exactly the same way as any other function in JavaScript. This means that any valid JavaScript code can be run inside the test function.

To run this test, simply navigate to the folder that contains the file squareRoot.test.js and enter the following command:

Una captura de pantalla de un celular con texto e imagen

Descripción generada automáticamente con confianza media

This will run all files that end in 'test.js' within that folder. The -c {} flag at the end is shorthand for ‘configuration’. We don't need any extra configuration, so we simply pass it an empty object.

If everything is working okay, it should produce the following output:

Hooray! This tells us that there was 1 test and it passed in a mere 2ms!

CRUNCHING SOME NUMBERS:

Texto

Descripción generada automáticamente

We have used the toEqual() match in this test. This is because we are testing an array.

This test says our factorsOf() function should return an array containing all the factors of 12 in order, when 12 is provided as an argument. If we run this test, we can see that it fails spectacularly:

Texto

Descripción generada automáticamente

To make things easier in this example, we're putting the code into the same file as the tests, but in reality you'd usually keep them in separate files.

Try running the test again:

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Oh dear, it still failed. This time, the failure message is a bit more specific. It says it was expecting the array [1,2,3,4,6,12] but received the array [1,2,3,4,6] ― the last number 12 is missing. Looking at our code, this is because the loop only continues while i < n. We need i to go all the way up to and including n, requiring just a small tweak to our code:

Texto

Descripción generada automáticamente

Our test passed, but this doesn’t mean we can stop there. There is still one more step of the TDD cycle: refactoring.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

The first test checks whether true is returned when a prime number (2) is provided as an argument, and another to check that true is not returned if a non-prime number (10) is given as an argument. These tests use the toBe() matcher to check if the result is true. Note the nice use of negation using the not matcher (although we should probably be checking if it’s false because this test will pass if anything but true is returned).

Our library of functions is growing! The next step is to again refactor our code. It’s a bit brittle at the moment, because both functions accept negative and non-integer values, neither of which are prime. They also allow non-numerical arguments to be provided. It turns out that the factorsOf() function fails silently and returns an empty array if any of these are passed to it. It would be better to throw an exception to indicate that an incorrect argument has been used. Let’s create some tests to check that this happens. Add the following tests to the numberCruncher.test.js file:

Una captura de pantalla con la imagen de una pantalla

Descripción generada automáticamente

These tests all use the toThrow() method to check that an exception has been thrown if the wrong data is entered as an argument.

While we’re at it, we can add some extra tests so the isPrime() function also deals with any incorrect arguments. No exceptions are necessary in these cases; non-numerical data, negative numbers and non-integers are simply not prime, so the function should just return false. Add the following code to the bottom of the numberCruncher.test.js file:

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media