Testing and Debugging

Errors, Exceptions, and Warnings

Errors are usually:

System Error- problem with the system or external devices that the program is interacting with

Programmer error – incorrect syntax or faulty logic; maybe a typo

User error – User entered data incorrectly that that program can't handle

Exceptions

An error that produces a return value that can then be used ny the program to deal with the error.

Stack Traces

a sequence of functions or method calls that lead to the point where the error occurred.

Warning

occurs if there is an error in the code that isn't enough to cause the program to crash.

Importance of Testing and debugging

It is good to ensure that the code we write fails loudly and not silently in the background so errors can be caught and fixed quickly. Goal to fail gracefully so user experience is not affected if possible.

Strict Mode

Introduced in ECMAScript 5 it produces more exceptions and warnings and doesn't allow use of some deprecated features. Can help improve clarity and speed and will throw exceptions if sloppy code is used. Sloppy Mode is not using strict mode. To use add this string to the first line of JavaScript file or inside a function if wanted inside a function:

'use strict';

Linting Tools

JS Lint, JS Hint, and ES Lint can be used to test the quality of code beyond strict mode. They highlight sloppy programming or syntax errors. Used for enforcing a programming style guide that is useful when working in a team. Can add a s a text-editor plug in or an online linting tool. Also install software using npm.

Feature Detection

Used to determine browser support for a feature. Use an if statement to check if an object or method exists before trying to call:

```
if (window.holoDeck) {
        virtualReality.activate();
}
```

Feature detection guarantees that the method is only called if it actually exists and fails gracefully, without any exceptions being thrown, if the method doesn't exist.

Modernizr library offers easy implementation for feature detection. Also Can I Use? is helpful.

Browser sniffing- old school way for checking browser support. Not recommended.

Debugging in the Browser

The origin of a bug is not always apparent so we may need to run through a program to see what is happening a different stages. **Breakpoints** are useful and they halt the progress of the code and let us view the value of different variables at that point in the program.

Options for debugging JavaScript in the browser:

alert() method- show a dialog at certain points in the code. Stops a program from running until OK is clicked. Allows us to add break points.

```
function amIOldEnough(age){
  if (age = 12) {
    alert(age);
    return 'No, sorry.';
  } else if (age < 18) {
    return 'Only if you are accompanied by an adult.';
  }
  else {
    return 'Yep, come on in!';
  }
}
```

Using the console- console object provides methods for logging info and debugging.

> console.log() method – log the value of variables a t different stages of the program. Does not stop execution of program.

```
function amIOldEnough(age){
  console.log(age);
    if (age < 12) {
    console.log(`In the if with ${age}`);
    return 'No, sorry.';
    } else if (age < 18) {
    console.log(`In the else-if with ${age}`);
    return 'Only if you are accompanied by an adult.';
    } else {
    console.log(`In the else with ${age}`);
    return 'Yep, come on in!';
  }
}
```

Debugging tools- many browsers also have debugging tool that allows you to set breakpoints in code that will pause it at certain points.

Debugger keyword creates a breakpoint in your code that will pause the execution of the code. Program can be restarted by clicking on the ‘play’ button:

```
function amIOldEnough(age){
  debugger;
    if (age < 12) {
    debugger;
    return 'No, sorry.';
    } else if (age < 18) {
        debugger;

        return 'Only if you are accompanied by an adult.';

        } else {

        debugger;

        return 'Yep, come on in!';

    }

}


amIOldEnough(16);
```

**remove debugger command before shipping code or will appear to freeze when people use it.

Error Objects- can be used using a constructor function:

const error = new Error();

Parameter can be used as the error message:

const error = new Error(‘Oops, something went wrong’);

7 more error objects used for specific errors:

eval(), RangeError, ReferenceError, SyntaxError, TypeError, URIError, InternalError.

Properties for error objects:

name, message, stack

Throwing Exceptions

You can throw your own exceptions using the throw statement. Allows problems to be highlighted and dealt with. Best practice to throw an error object. This can then be caught in a catch block.

throw new Error('Something has gone badly wrong!');

Example: if someone tries to use the Math.sqrt() method using a negative number:

```
function squareRoot(number) {
    'use strict';

    if (number < 0) {

        throw new RangeError('You can't find the square root of negative numbers')

    }

    return Math.sqrt(number);

};
```

Exception Handling

When exception occurs, program terminates with error message. Ideal for development, not for production because it appears the program has crashed.

Can handle exceptions gracefully by catching the error. Can be hidden from users, but still identified. Can deal with error or ignore for time being.

Try, catch, and finally

If we think a piece of code will result in an exception, we can wrap it in a try block. Will run code inside block as normal, but will pass an error object that is thrown onto a catch block:

```
function imaginarySquareRoot(number) {

    'use strict';

    try {

        return String(squareRoot(number));

    } catch(error) {

        return squareRoot(-number)+'i';

    }

}
```

A finally block can be added to a catch block. This will always be executed after the try or catch block, whether an exception occurred or not:

```
function imaginarySquareRoot(number) {

    'use strict';

    let answer;

    try {
        answer = String(squareRoot(number));
    } catch(error) {
        answer = squareRoot(-number)+"i";
    } finally {
        return `+ or - ${answer}`;
    }
}
```

Tests

Tests allow you to identify errors early on and code will be less brittle. Can be a function that tests a piece of code.:

```
function itSquareRoots4() {
    return squareRoot(4) === 2;
}
```

Here we're comparing the result ofsquareRoot(4)with the number2. This will returntrueif our function works as expected, which it does:

Copy

```
itSquareRoots4();
<< true
```

Test-driven development

TDD process of writing test before any actual code. 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

Considered best practice, but not always used even by best programmers.

Testing Frameworks

Provide a structure to write meaningful tests and then run them. Many different frameworks.

Jest- TDD Framework created by Facebook.