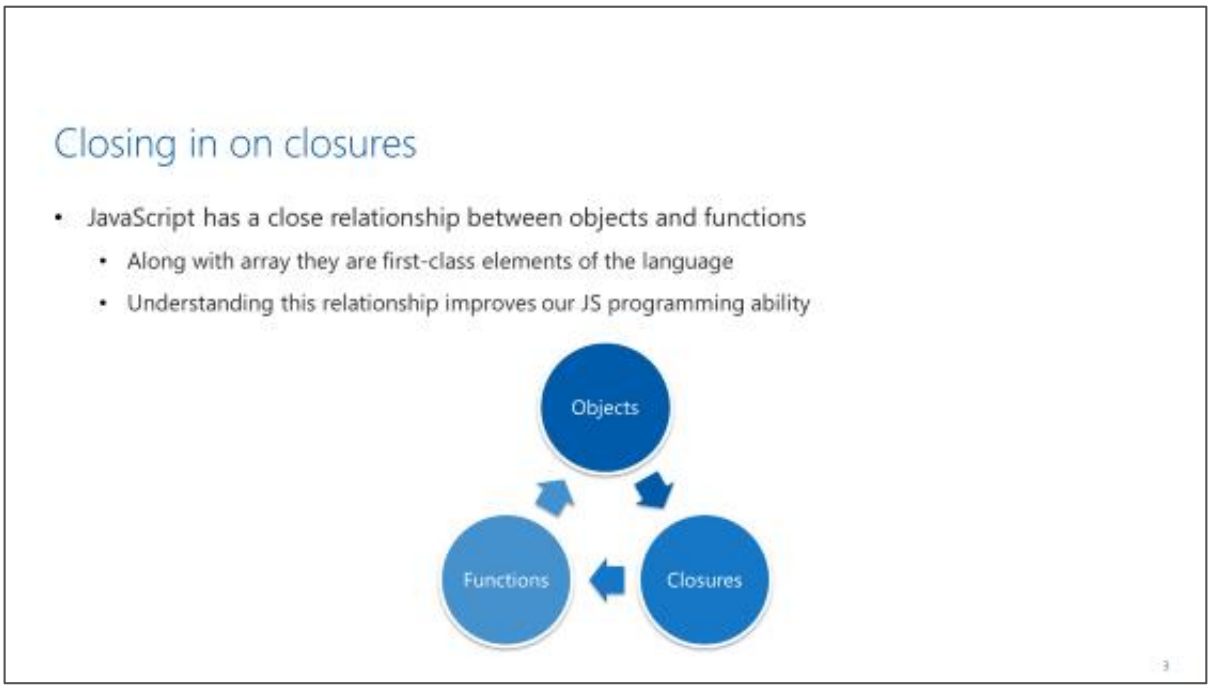
Closures & self-executing functions

PROGRAMMING WITH JAVASCRIPT

Introduction

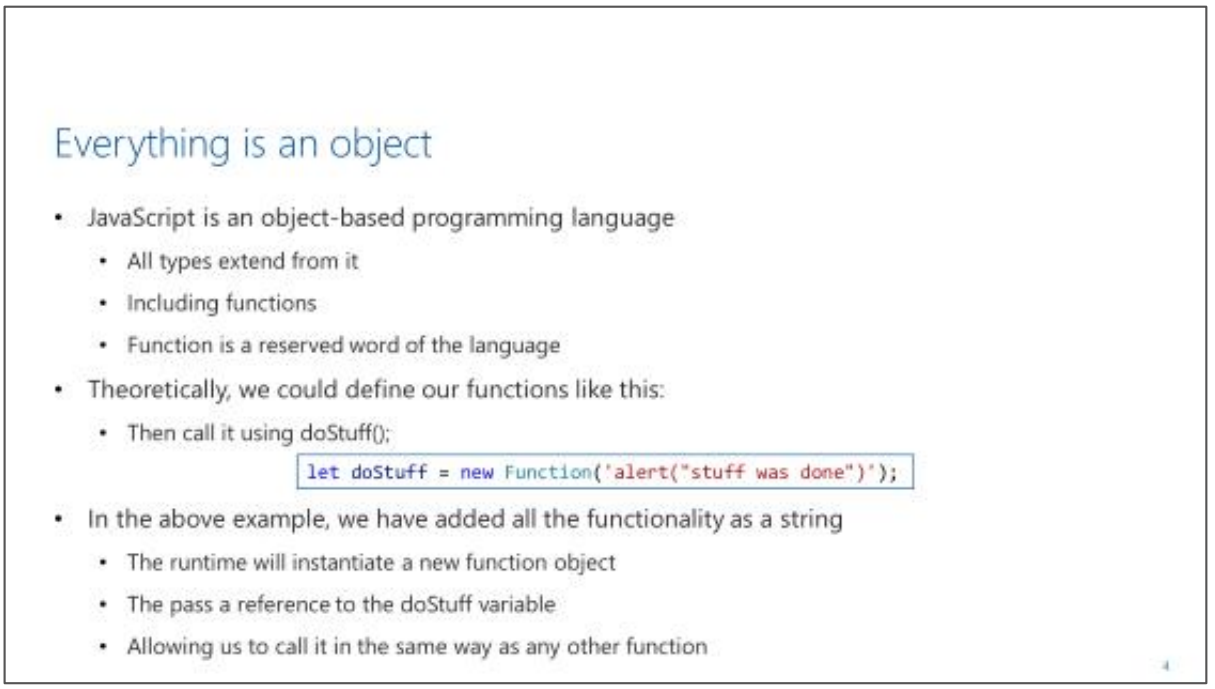
- Advanced function design
  - Closures
    - What are closures and how do they work?
    - Using closures to simplify development
  - Self-executing functions
- Robust JavaScript design
  - Strict mode

2



JavaScript consists of a close relationship between objects, functions – which, in JavaScript, are first-class elements – and closures. Understanding the strong relationship between these three concepts vastly improves our JavaScript programming ability, giving us a strong foundation for any type of application development.

Traditionally, closures have been a feature of purely functional programming languages. Having them cross over into mainstream development has been particularly encouraging, and enlightening. It's not uncommon to find closures permeating JavaScript libraries, along with other advanced code bases, due to their ability to drastically simplify complex operations.



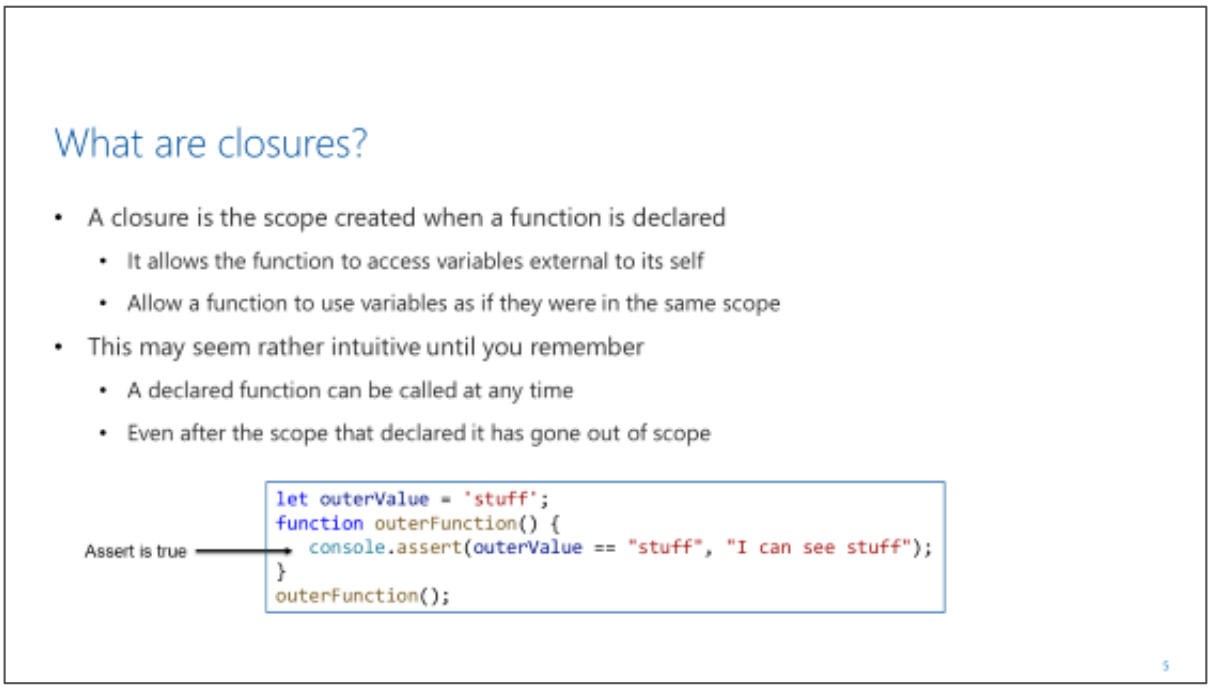
Objects are the building blocks of the JavaScript language. In fact, JavaScript is defined as an *object based programming language*. Absolutely everything we work with in JavaScript has an object at its core. Consider the following code:

```
let x = 5;
let y = new Number(5);
```

Both code implementations create an object of type number and the variable then receives a memory reference to that object. It is important to remember that JavaScript variables are simply pointers to this object.

This concept extends to functions. In the code block above, we see another way of creating a function. The **new** keyword instantiates a function, with the logic is passed in as a string. **doStuff** receives a reference to the function. As long as the variable **doStuff** remains in scope, the function remains available.

N.B. THIS IS A THEORETICAL APPROACH! Never implement functions like this! Later in the course your instructor will show you a pattern called self-executing functions that create a security vulnerability of incredible risk.



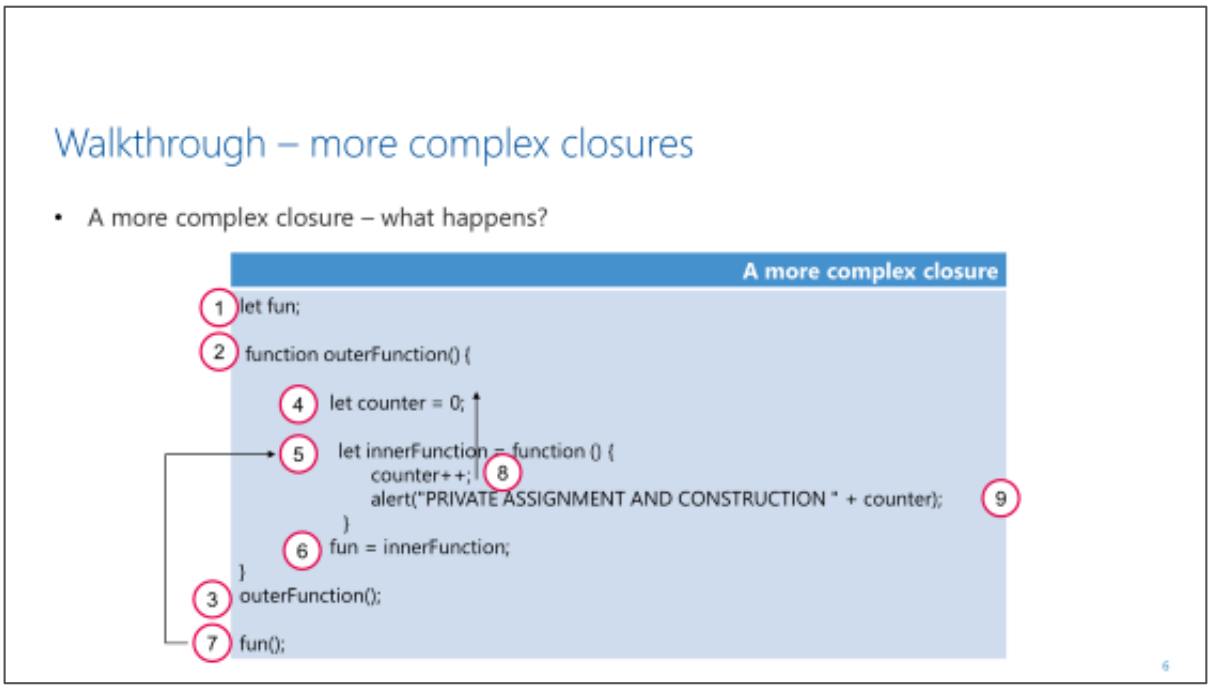
In this code example, we declare a variable **outerValue** and a function **outerFunction** in the same scope. Afterwards, we cause the function to execute. The assert condition returns true as we are able to see the **outerValue** variable.

Closures allow a function to access all the variables as well as other functions that are declared in the same scope that the function itself is declared.

That may seem rather intuitive until you remember that a declared function can be called at any later time, even after the scope in which it was declared has gone away.

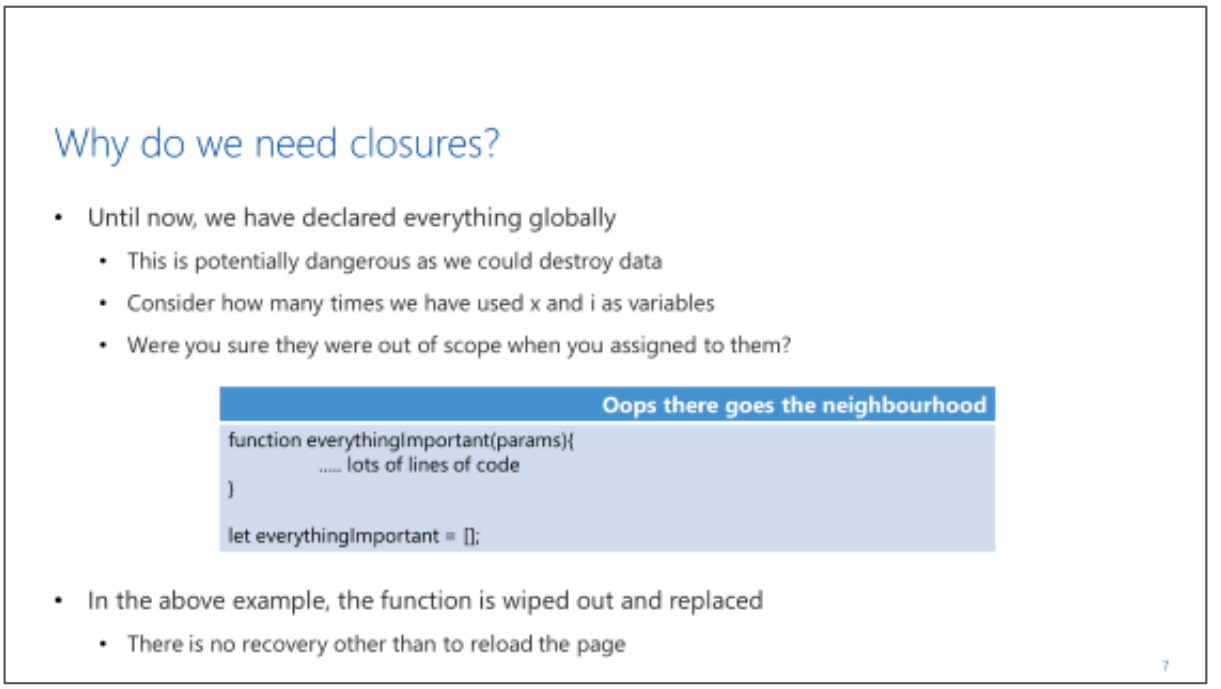
Remembering that every variable, object and function is contained within an object. Both the outer value and the outer function are declared in global scope (window object); that, as long as the page is loaded, that scope never goes away.

So, in this scenario we have created two global resources. What would be the outcome of the code **var outerFunction = "hello";** ?



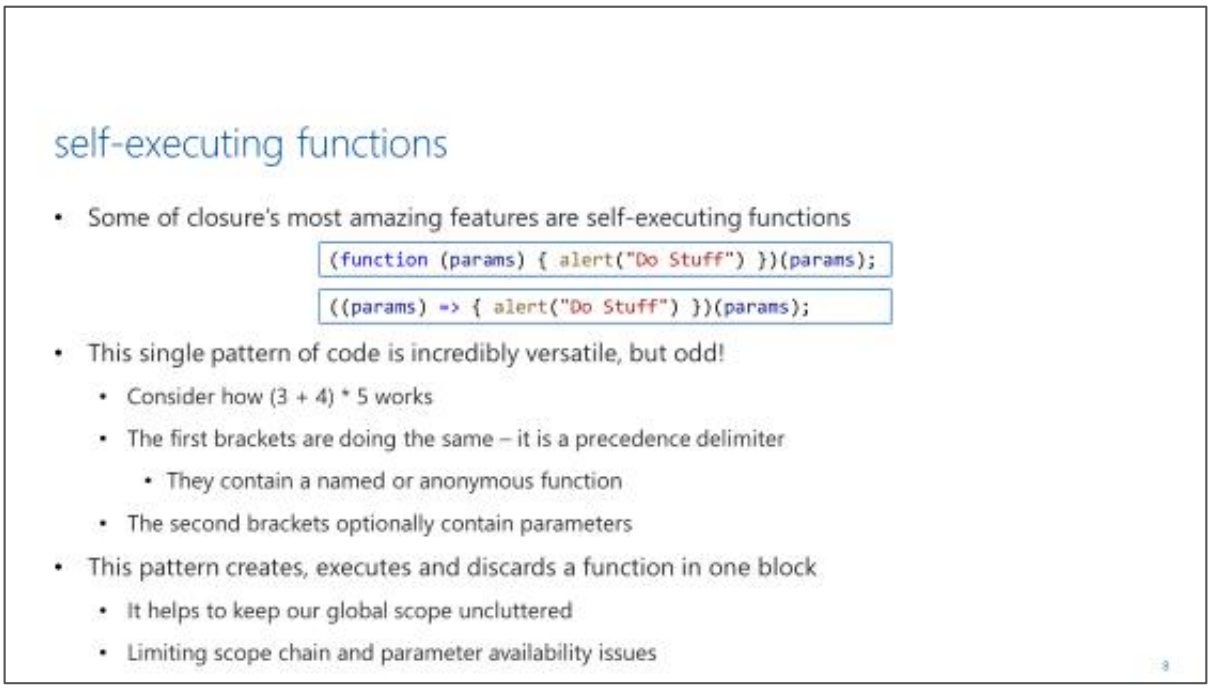
We are going to explore closures in a little more depth here:

1. We declare a variable called fun – what is its current value?
2. We declare a function called outerFunction – remember that functions are objects.
3. **outerFunction** is called by the global stack.
4. Within outerFunction's body, a number primitive is created with an id of **counter.**
5. A function called innerFunction is created and a reference to the heap object is passed to the innerFunction variable.
6. Fun copies the value of innerFunction, that value is a memory pointer to the innerFunction function.
7. The function now exits, in most scenarios 'counter', and 'innerFunction' go out of scope and would be destroyed as they are no longer needed. In this instance, that is not the case. **fun** now contains the address of the innerFunction object. For as long as fun is in scope the object created on the heap has a valid memory reference. As a result, it remains alive.
8. So the call stack now moves to the function we have created and invokes it. In turn, this function adds one to the value of counter. This is where we see the real power of closures. Counter was not released either because innerFunction holds a reference to it.
9. As a result, we have an enclosed counter. Fun knows about innerFunction and innerFunction knows about counter. While fun remains in scope so do innerFunction and counter.



The case above talks about the danger, that accidentally you can override a function or variable and cause a massive amount of damage to your code. We are also being very wasteful. Consider how many times you have used **document.querySelectorAll** to create an array of objects ready to hook up to events as an example. What do we use that array for as long as our page is live? What an amazing waste of memory!

Closures are part of the answer, but we would still have a vulnerability as the function that encloses it could still be wiped out. We can solve that problem with self-executing functions.



This single pattern of code is incredibly versatile and ends up giving the JavaScript language a huge improvement in power and performance. Its syntax, with all those braces and parentheses, may seem a little strange, so let's deconstruct what's going on step by step.

First, let's ignore the contexts of the first set of parentheses, and examine the construct:

(…)()

We know that we can call any function using the functionName() syntax, but in place of the function name, we can use any expression that references a function instance. That's why we could call a function from a variable that refers to a function using the variableName() syntax. As in other expressions, if we want an operator (in this case the function call operator () to be applied to an entire expression, we'd enclose that expression in a set of parentheses.

So in (…)(), the first set of parentheses is merely a set of delimiters enclosing an expression while the second set is an operator.

The result of this code is an expression that creates, executes, and discards a function all in the same statement. Additionally, since we're dealing with a function that can have a closure like any other, we also have access to all outside variables in the same scope as the statement.

Using self-executing functions

- Using self-executing, we can build up interesting enclosures
  - Examine the following code

```
(() => {
    let numClicks = 0;
    document.addEventListener("click", function () {
        alert(++numClicks);
    });
})();
```

  - The function is executed immediately
  - The click handler is bound right away
  - numClicks is enclosed in a document level event listener
  - It is now a private variable

9

Using immediate functions, we can start to build up interesting enclosures for our work. Because the function is executed immediately, and all the variables inside of it are confined to its inner scope, we can use it to create a temporary scope within which our state can be maintained.

Remember variables in JavaScript are scoped to the function within which they were defined. By creating a temporary function, we can use this to our advantage and create a temporary scope for our variables to live in.

As the immediate function is executed immediately (hence its name), the click handler is also bound right away. Additionally, note that a closure is created allowing the **numClicks** variable to persist with the handler but nowhere else. This is the most common way in which immediate functions are used, just as a simple self-contained wrapper for functionality.

Parameterising self-executing functions

- The last example was only ever good for the document object
  - We should parameterise it so the pattern can be reused
  - Use the second set of brackets in the function to do this

```
(el => {
    let numClicks = 0;
    el.addEventListener("click", function () {
        alert(++numClicks);
    });
})(document.getElementById('d1'));
```

- We are passing the result of getElementById into the function
  - No reference to anything pollutes the global scope

10

The anonymous function now takes parameters. Use the second set of brackets to pass parameters into the global function as you normally would.

Block Statement

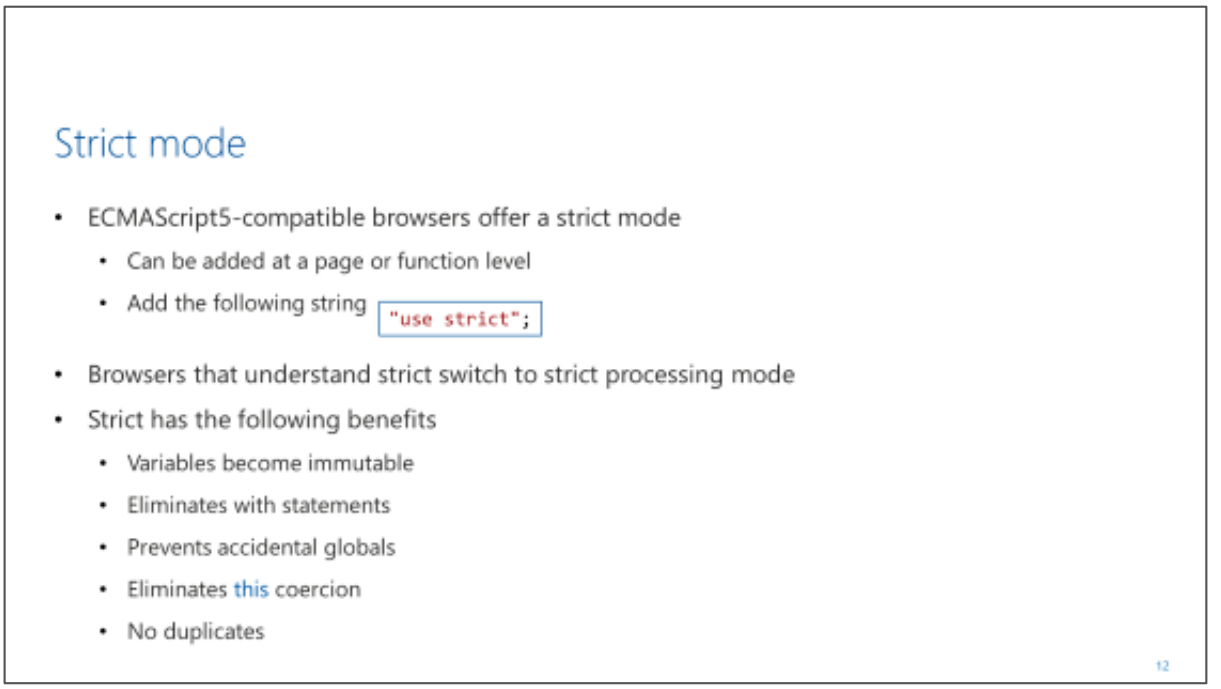
- While IIFEs still have a role to play in JavaScript development, and Closures are an inescapable core tenant of the language that is very useful to use and critical to understand, block statements combined with the let keyword give us another (simpler) avenue for encapsulating our variables

```
{
    let el = button;
    let numClicks = 0;
    el.addEventListener("click", evt => alert(++numClicks));
}
```

- let ensures that these variables are scoped to this block, whilst not being contained within a function means it is immediately evaluated.

11

The anonymous function now takes parameters. Use the second set of brackets to pass parameters into the global function as you normally would.



ECMAScript 5 introduced strict mode to JavaScript. The intent is to allow developers to opt in to a "better" version of JavaScript, where some of the most common and egregious errors are handled differently.

The goals of strict mode is to allow for the faster debugging of issues. The best way to help developers debug is to throw errors when certain patterns occur, rather than silently failing or behaving strangely (which JavaScript does today outside of strict mode). Strict mode code throws far more errors, and that's a good thing, because it quickly calls to attention things that should be fixed immediately.

Review

- Closures
  - Closures give us a great way of managing scope
  - Avoids extending the window object unnecessarily
- Self-executing functions
  - A great side effect of the closure pattern
  - The key to advanced JavaScript
- Strict mode
  - Learn to love it

13

Exercise

- Building a self-executing event association
- Evaluating strict mode

14