Types and operators

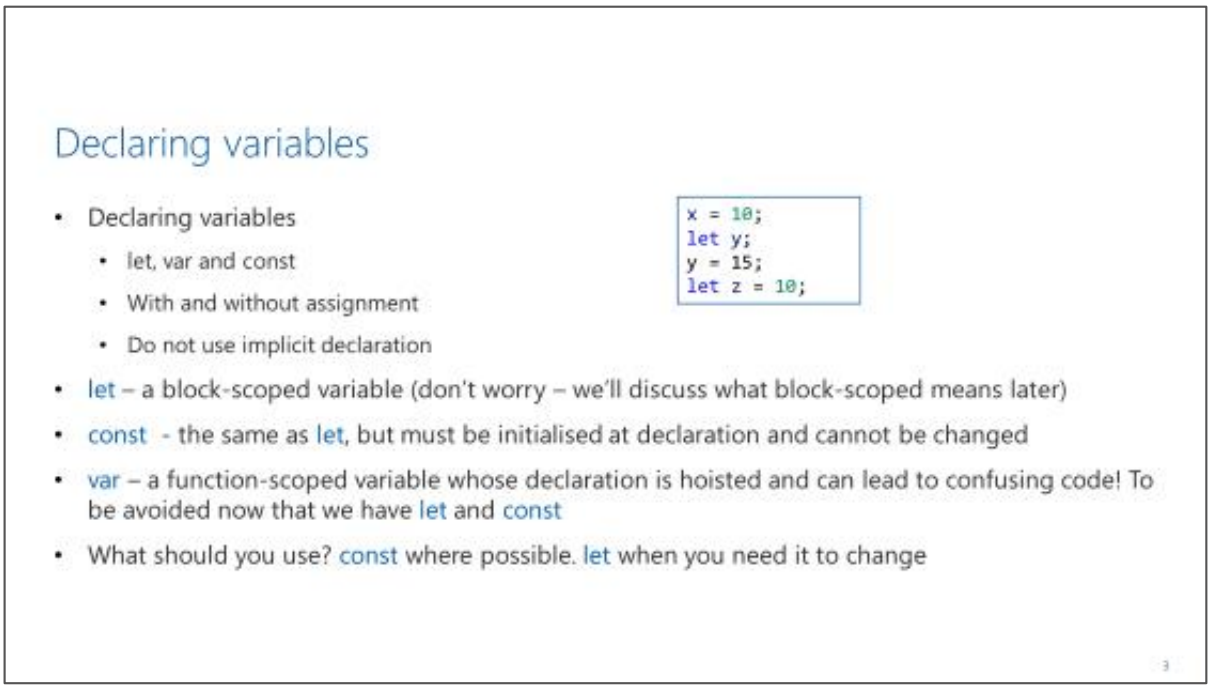
PROGRAMMING WITH JAVASCRIPT

Introduction

In this module, you will learn to:

- Declare variables
- Understand types
  - Primitive types
    - Strings
    - Numbers
    - Booleans
    - Undefined
    - Nulls
    - Symbol
  - Reference types
- Operators
  - Using operators
  - Type conversion

2



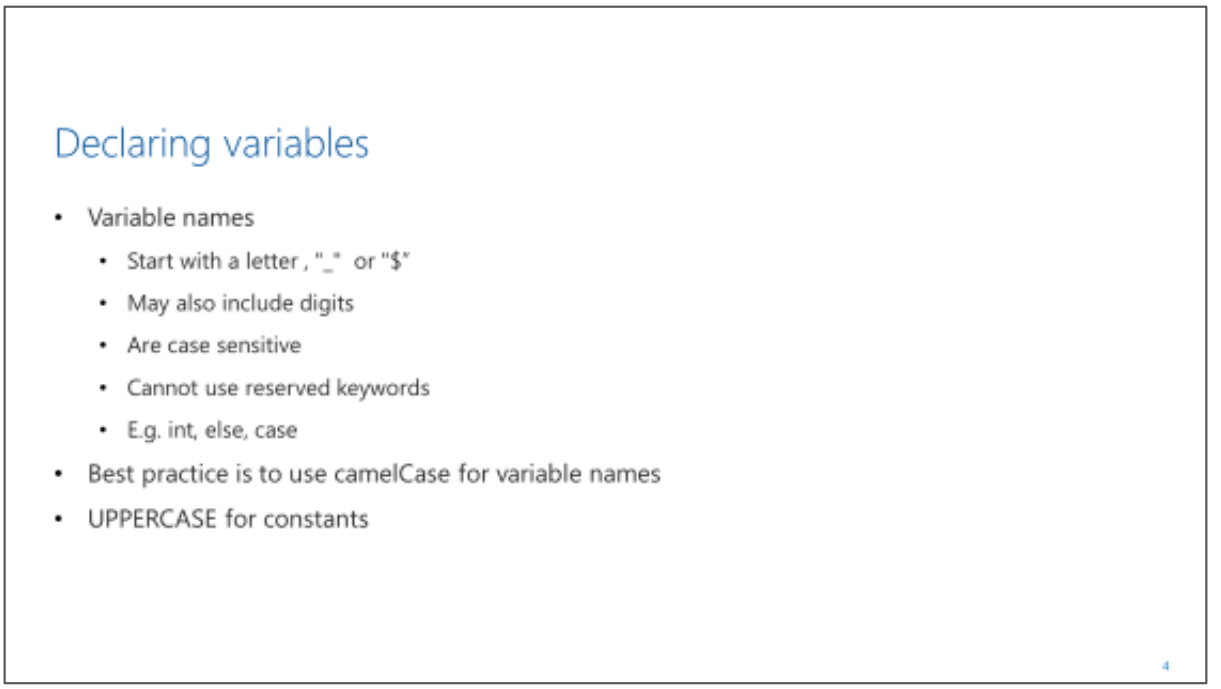
Variables in JavaScript should be declared with the **var** operator followed by the variable name. If you look above, you will see the variable **x** does not follow this rule. It is valid, just terribly bad practice and has an impact on variable scope as we will examine later.

The second variable, **y**, is valid but currently uninitialised. This is perfectly valid if somewhat dangerous. A JavaScript variable can be assigned at a later point and its type will be set. In fact, that is one of the most interesting things about JavaScript variables – they can change type part way through their life.

We must also be mindful of variable names and try to follow conventions. Variables can start with a $ or an _ but these are reserved for special situations that we will discuss later in the course. For now, our variables will be alphanumeric-based. Most variables will follow a specific naming convention:

- camelCasing – used for variable names
- PascalCasing – used for objects (discussed later)
- sHungarianNotation – where the datatype of the variable prefixes the variable name. This can be useful in JavaScript, when the variable is not set with an explicit type making the code more human readable

It is usually best to have a human-readable variable name, but more experienced developers attempting to obfuscate their code go for terse names.



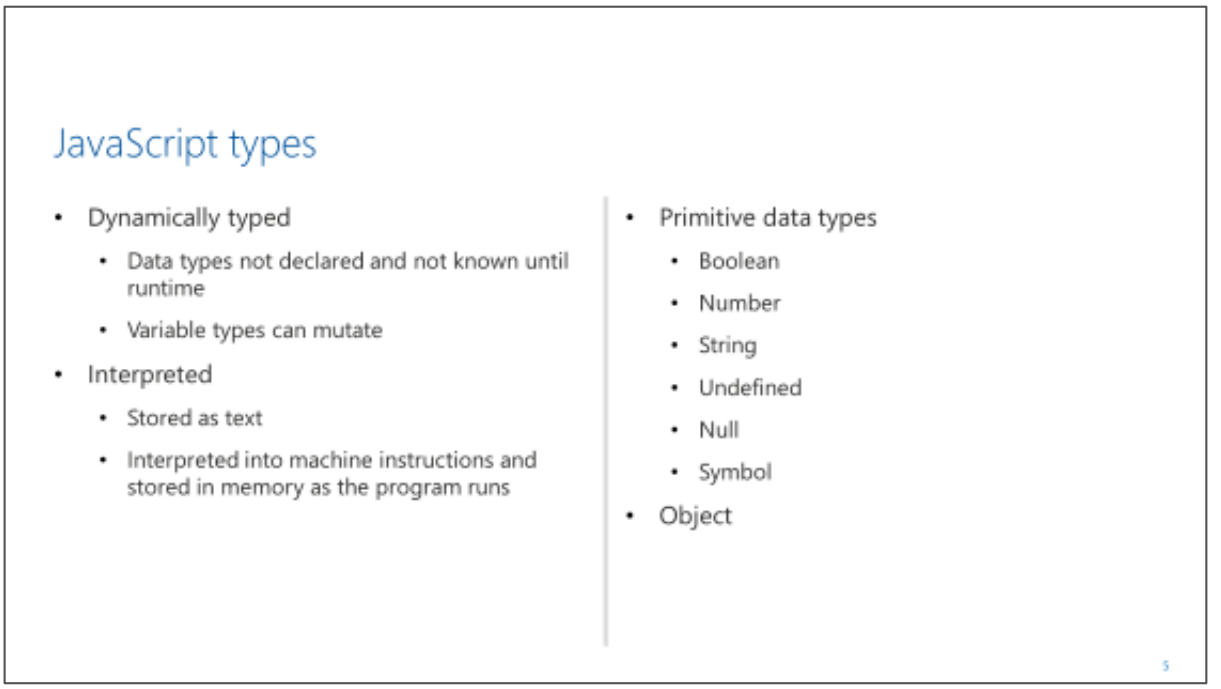
Variables in JavaScript should be declared with the **var** operator followed by the variable name. If you look above, you will see the variable **x** does not follow this rule. It is valid, just terribly bad practice and has an impact on variable scope as we will examine later.

The second variable, **y,** is valid but currently uninitialised. This is perfectly valid if somewhat dangerous. A JavaScript variable can be assigned at a later point and its type will be set. In fact, that is one of the most interesting things about JavaScript variables – they can change type part way through their life.

We must also be mindful of variable names and try to follow conventions. Variables can start with a $ or an _ but these are reserved for special situations that we will discuss later in the course. For now, our variables will be alphanumeric-based. Most variables will follow a specific naming convention:

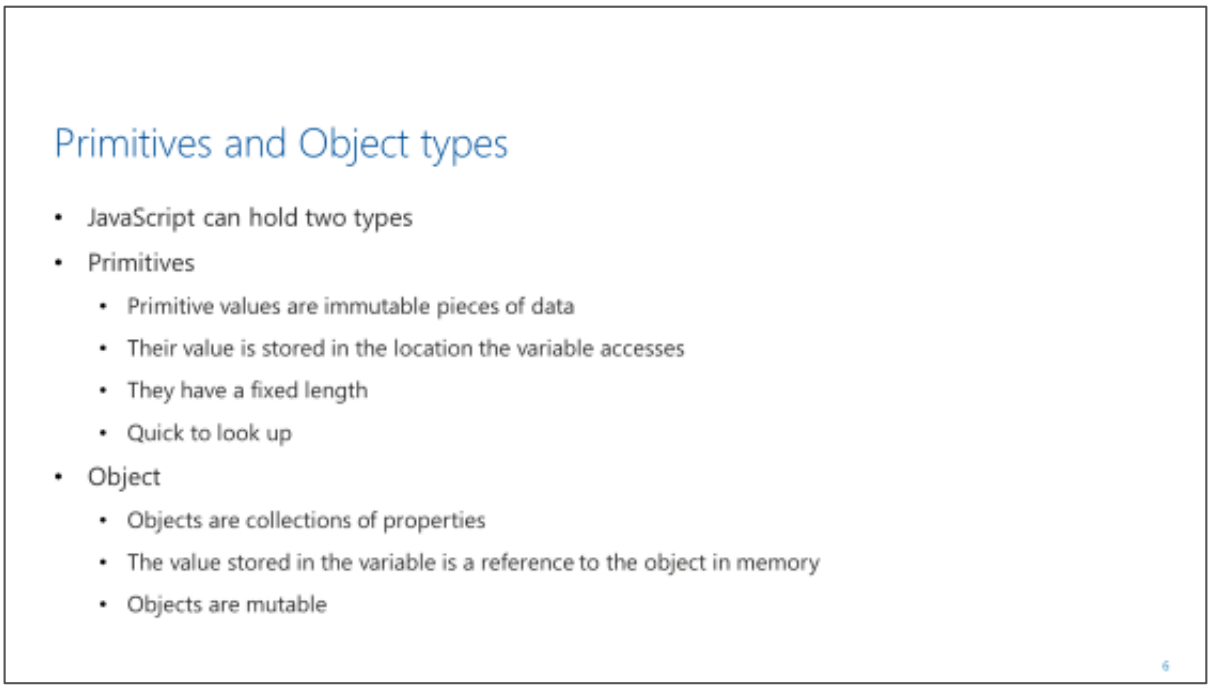
- camelCasing – used for variable names
- PascalCasing – used for objects (discussed later)
- sHungarianNotation – where the datatype of the variable prefixes the variable name. This can be useful in JavaScript when the variable is not set with an explicit type making the code more human readable

It is usually best to have a human-readable variable name, but more experienced developers attempting to obfuscate their code go for terse names.



As we have discussed, variables in JavaScript can be dynamically created and assigned variables at run time. The JavaScript interpreter in the browser parses the instructions and creates a memory location holding the data.

JavaScript provides a limited number of types that we will explore over the next few pages. If you are used to more full-fat programming languages, like Java or C#, you may be wondering where the doubles and precision numeric types are among others. You simply do not have them in JavaScript and that makes life a little more interesting.

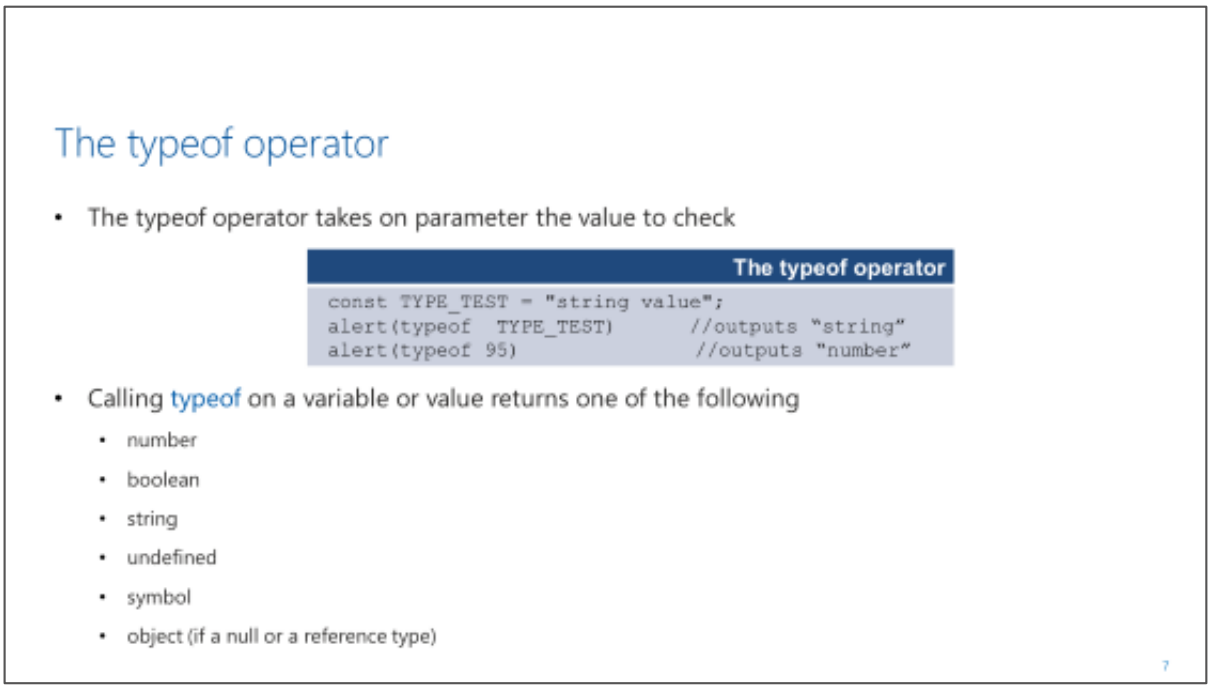


When we create a variable, you are actually instructing the JavaScript interpreter to grab hold of a location in memory. When a value is assigned to a variable, the JavaScript interpreter must decide if it is a primitive or reference value. To do this, the interpreter tries to decide if the value is one of the primitive types:

- Undefined
- Null
- Boolean
- Number
- String
- Symbol

Each primitive type takes up a fixed amount of space. It can be stored in a small memory area, known as the stack. Primitive values are simple pieces of data that are stored on the stack. This means their value is stored directly in the location that the variable accesses.

If the value is a reference, then the space is allocated on the heap. A reference value can vary in size, so placing it on the stack would reduce the speed of variable lookup. Instead, the value placed in the variable's stack space is the address of a location in the heap where the object is stored.



`typeof` is an incredibly useful operator in the weakly-typed JavaScript language. It allows us to evaluate the type of a variable or value. It is extremely useful with primitives but null and reference types always evaluate to the generic object type.

The undefined type

- A variable that has been declared but not initialised is undefined

The undefined type

```
let age;
console.log(typeof age);  //returns undefined
```

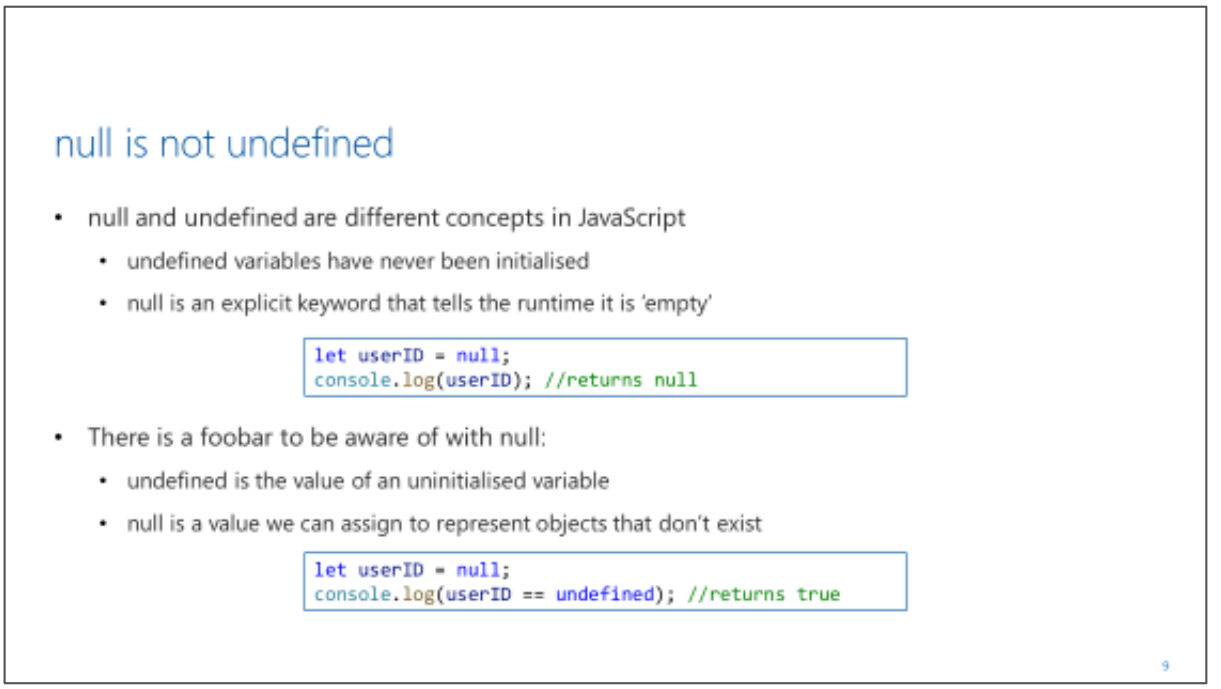
- A variable that has not been declared will also be undefined
  - The typeof operator does not distinguish between the two

The undefined type

```
//let boom;
console.log(typeof boom);  //returns undefined
```

  - It is a good idea to initialise variables when you declare them

A variable that has not been initialised is undefined. As we will see, undefined and null can have a lot of similar properties, if we were to use them in operator statements, so we should preferably give a variable a type even if that type is null.



NULL is very useful and carries a much more implicit meaning than an undefined and uninitialised variable. If the value of a variable is to be set to a reference object such as an array, be careful in your code because a variable that is undefined will evaluate the same as null.

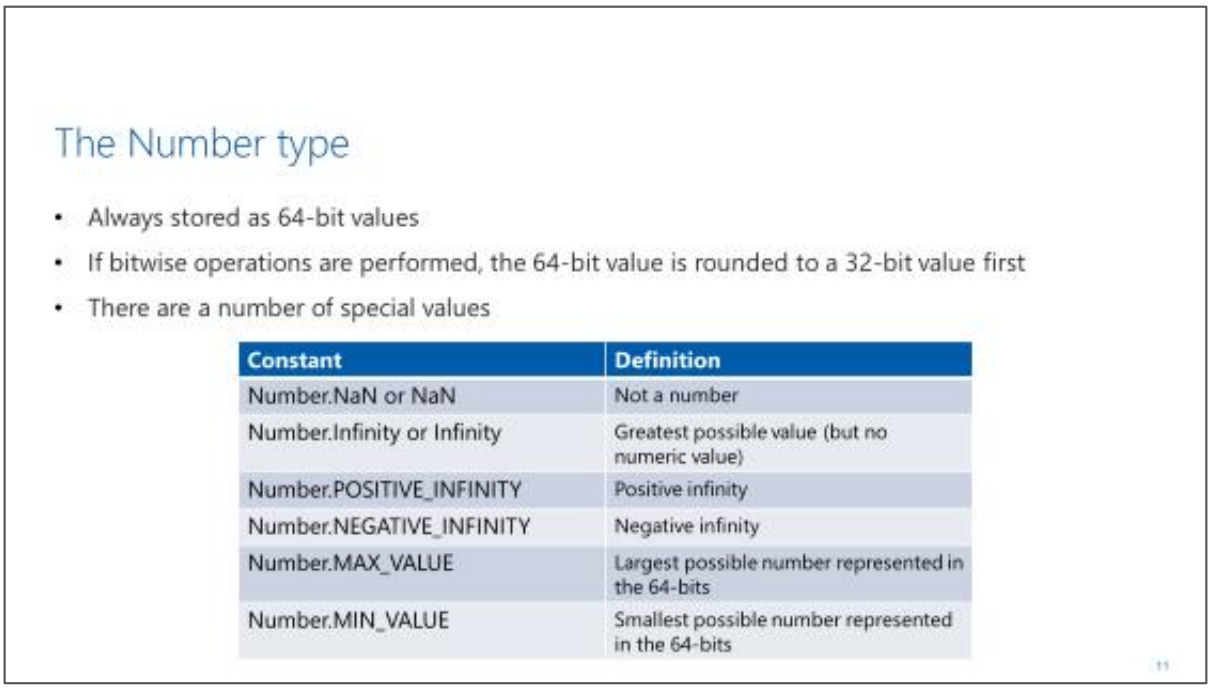
The Boolean type

- Boolean can hold two values – true and false
- These are reserved words in the language:

```
var loggedOn = false;
console.log(loggedOn); //returns false
```

- When evaluated against numbers, you can run into issues
  - false is evaluated as 0
  - true can be evaluated to 1
  - Vice-versa as well

10



In JavaScript, numbers are always stored as 64-bit values. However, if you perform a bitwise operation then the value is truncated to 32-bits. This is important to remember when you require a large bit field. During regular arithmetic, division can always produce a result with fractional parts.

There are a number of special values that are listed above and a series of object functions/methods noted below:

Method	Description
`toExponential(n)`	Converts a number into an exponential notation
`toFixed(n)`	Formats a number with n numbers of digits after the decimal
`toPrecision(n)`	Formats a number to x length
`toString(n)`	Converts a Number object to a string
`valueOf()`	Returns the primitive value of a Number object

The String type

- Immutable series of zero or more Unicode characters
  - Modification produces a new string
  - Can use single (') or double quotes (") or backticks (`)
  - Primitive and not a reference type
- String concatenation is expensive
- Back-slash (\) used for escaping special characters

Escape	Output
\'	'
\"	"
\\	\
\b	Backspace
\t	Tab
\n	Newline
\r	Carriage return
\f	Form feed
\ddd	Octal sequence
\xdd	2-digit hex sequence
\udddd	Unicode sequence (4-hex digits)

12

As with many other languages, strings in JavaScript are an immutable sequence of zero or more Unicode characters. If we modify a string, for example by concatenation, then a new string is allocated. We can use single or double quotes or backticks in JavaScript.

As is common with many languages, certain special characters must be represented as an escape sequence a back-slash followed either by the character itself, by a significant letter or by a code as shown in the table above.

String Concatenation and Interpolation

- Adding 2 (or more strings) is an expensive operation due to the memory manipulation required
- To concatenate a string the + operator is used

```
let str1 = "5 + 3 = ";
let value = 5 + 3;
let str2 = str1 + value
console.log(str2); // 5 + 3 = 8
```

- Template literals (introduced in ES2015) allow for strings to be declared with JavaScript expressions that are evaluated immediately using ${} notation

```
let str2 = `5 + 3 = ${5 + 3}`;
console.log(str2); // 5 + 3 = 8
```

13

String functions

- The String type has string manipulation methods, including

Method	Description
indexOf()	Returns the first occurrence of a character in a string
charAt()	Returns the character at the specified index
toUpperCase()	Converts a string to uppercase letters

- Method is called against the string variable

```
let str = "Hello world, welcome to the universe.";
let n = str.indexOf("welcome");
```

  - Where n will be a number with a value of 13

14

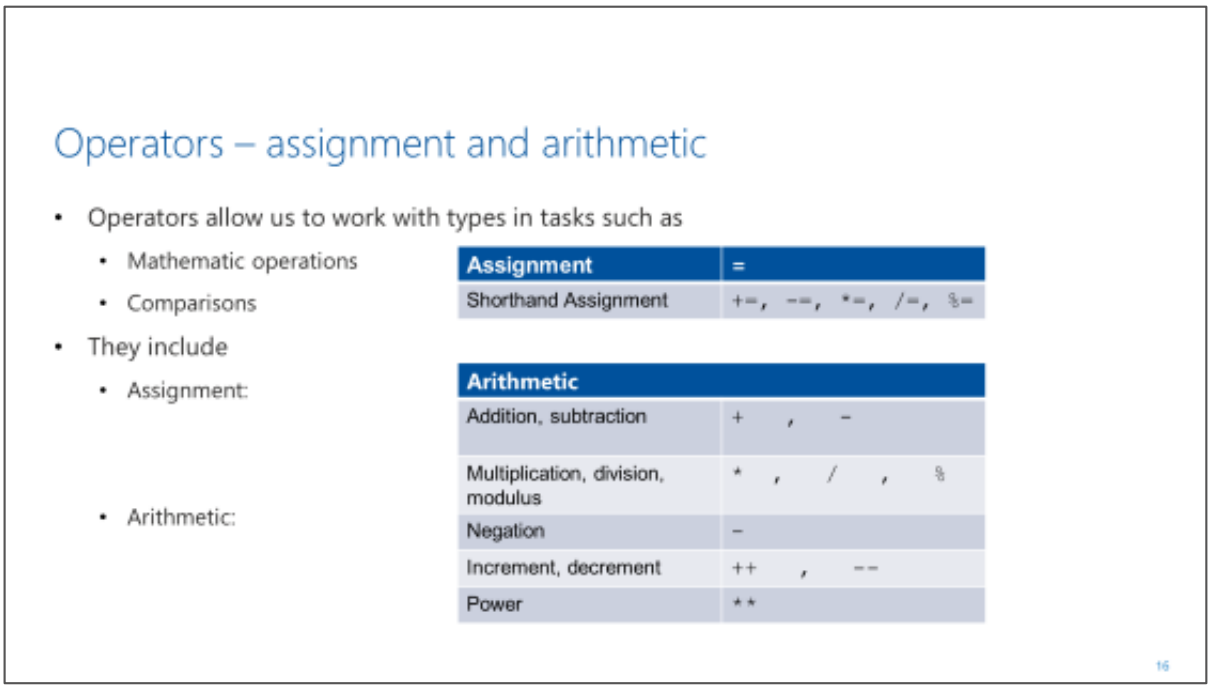
Further methods:

Method	Description
`charCodeAt()`	Returns the unicode of the character at the specified index
`concat()`	Joins strings and returns a copy of the joined string
`fromCharCode()`	Converts unicode values to characters
`lastIndexOf()`	Returns the position of last found occurrence from a string
`slice()`	Extracts part of a string and returns a new string
`split()`	Splits a string into an array of substrings
`substr()`	Extracts from a string specifying a start position and number of characters
`substring()`	Extracts from a string between two specified indices
`toLowerCase()`	Converts a string to lower case

Exercise 2 – part 1

- Exploring types
- Create variables of a number type
  - Using methods of the number object
- Creating variables of a string type
  - Using string functions to manipulate string values

15

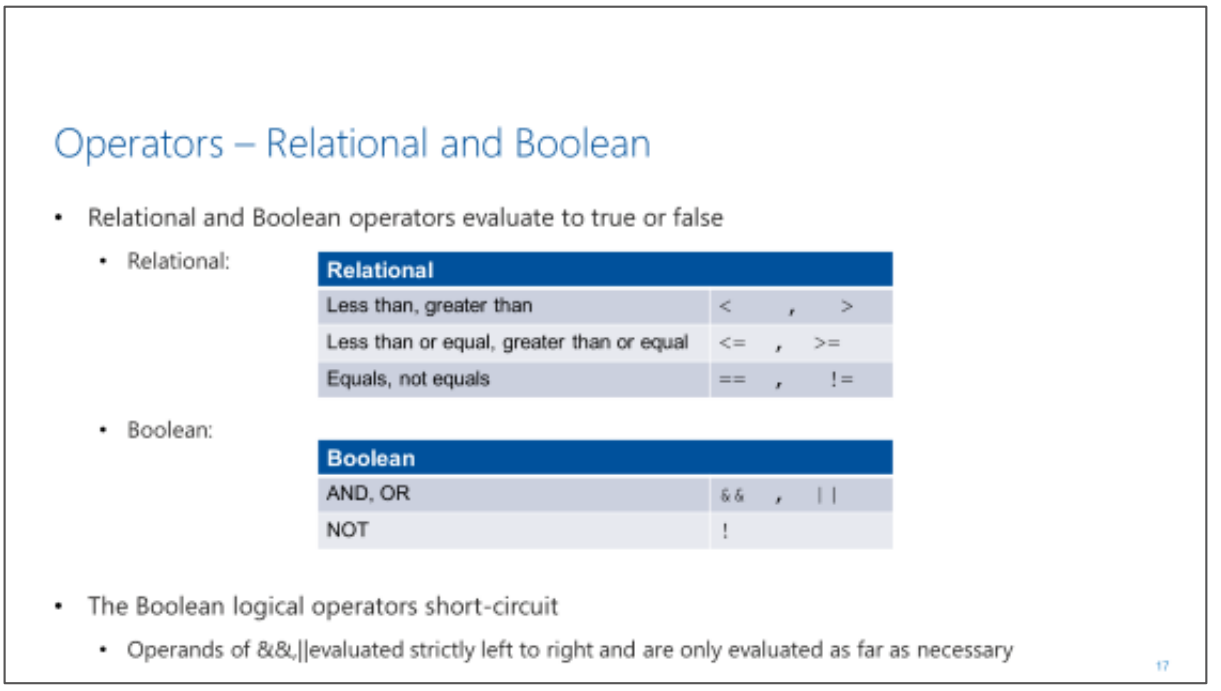


JavaScript has all the operators that you would expect for a modern language; in general, they follow the same representation as first became popular in the C language.

In mathematic expressions, there is an order of precedence, e.g. `5 + 3 * 10` returns a value of 35 because the multiplication is dealt with before the addition.

The following piece of code uses some of the assignment operators to do the same thing. JavaScript programmers like to do things in as few characters as possible:

```
var x = 0;
x = x+ 1; //x is now 1
x+= 1; //x is now 2
x++; //x is now 3
X**2; //x is now 9
```



The Boolean AND and OR operators have 'short-circuit' evaluation. This means that when an expression involving them is evaluated, it is only evaluated as far as is necessary. For example, consider the expression:

```
if (exprA && exprB)
```

If exprA is false, then exprA && exprB must also be false, so there is sometimes no point evaluating exprB. exprB will only be evaluated if exprA is true; indeed, the && operator will simply return the value of exprB if exprA is true.

Type checking

- JavaScript is a loosely-typed language

```
let a = 2;
let b = "two";
let c = "2";
alert(typeof a);// alerts "number"
alert(typeof b);// alerts "string"
alert(typeof c);// alerts "string"
```

- JavaScript types can mutate and have unexpected results

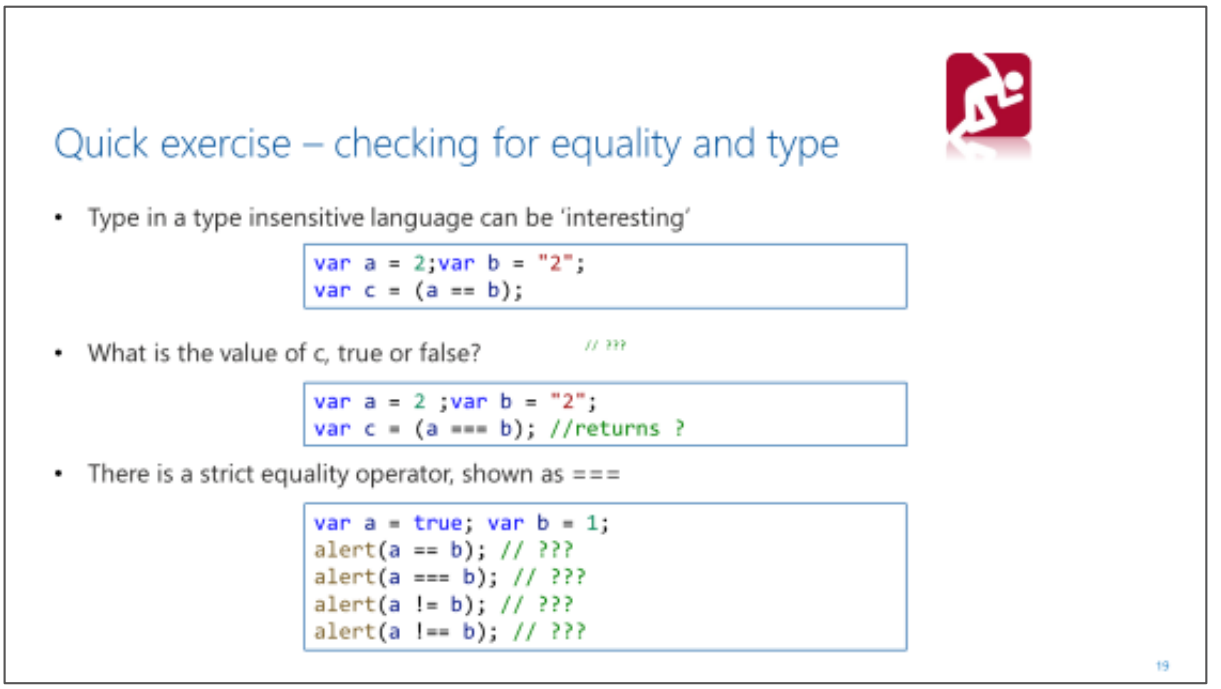
```
alert(a * a);// alerts 4
alert(a + b);// alerts 2two
alert(a * c);// alerts 4
alert(typeof (a * a));// alerts "number"
alert(typeof (a + b));// alerts "string"
alert(typeof (a * c));// alerts "number"
```

18

When we 'add' a string and a number using the + operator, JavaScript assumes we're trying to concatenate the two, so it creates a new string. It would appear to change the number's variable type to string. When we use the multiplication operator (*) though, JavaScript assumes that we want to treat the two variables as numbers.

The variable itself remains the same throughout, it's just treated differently. We can always explicitly tell JavaScript how we intend to treat a variable; but, if we don't, we need to understand just what JavaScript is doing for us. Here's another example:

```
alert(a + c);// alerts 22
alert(a + parseInt(c));// alerts 4
```



Two = signs together, ==, is known as the equality operator, and establishes a Boolean value. In our example, the variable will have a value of true, as JavaScript compares the values before and after the equality operator, and considers them to be equal. Using the equality operator, JavaScript pays no heed to the variable's type, and attempts to coerce the values to assess them.

Switch out the first equal sign for an exclamation mark, and you have yourself an inequality operator (!=). This operator will return false if the variables are equal, or true if they are not.

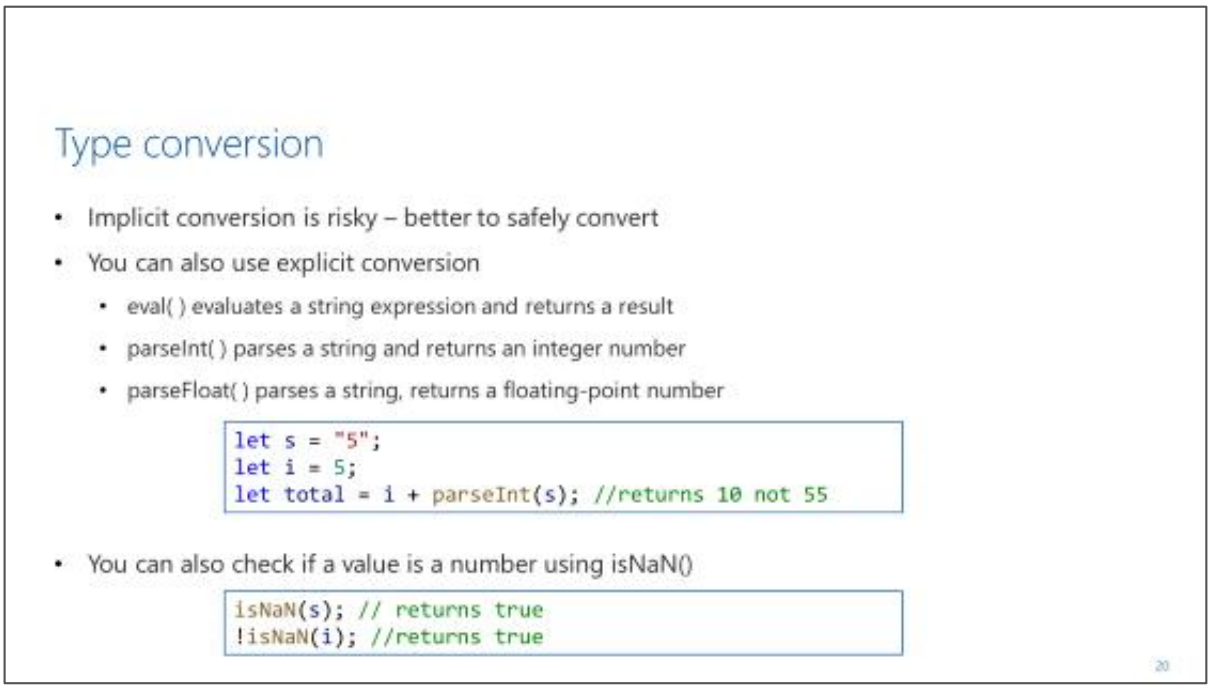
In JavaScript 1.3, the situation became even less simple, with the introduction of one further operator: the strict equality operator, shown as ===.

The strict equality operator differs from the equality operator, in that it pays strict attention to type as well as value when it assigns its Boolean. In the above case, d is set to false; while a and b both have a value of 2, they have different types.

And, as you might have guessed, where the inequality operator was paired with the equality operator, the strict equality operator has a corresponding strict inequality operator:

```
var f = (a !== b);
```

In this case, the variable will return true, as we know the two compared variables are of different types, though their values are similar.



As we discovered, type mismatching can cause some serious logical issues while working with JavaScript, and is often better to explicitly take control of type conversion. There are three key functions here:

`eval()` – commonly used to create string arrays. We will examine this function in more depth later in the course.

`parseInt()` – takes a value or variable that is not currently a number and tries to convert its value into a numeric type. Specifically, it is looking for numeric values and any decimal points. It preserves anything to the left of the decimal point, so:

parseInt(55.95) would return 55, note that no rounding has occurred

parseInt("55.95boom!") would also return 55

`parseFloat()` – works as per parseInt(), but preserves numeric values after the decimal point:

parseFloat(55.95) would return 55.95 as a number

parseFloat("55.95boom!") would also return 55.95 also

Both **parseInt** and **parseFloat** return a NaN error object if the conversion can not occur, which you can detect using the **isNan()** function.

Exercise 2 – part 2

- Exploring operators and types
- Arithmetic types
- Relational operators
- Assignment operations
- Type mismatching and conversion

21

Review

- Primitive variables
  - Value types
- Understand types
  - There are six primitive types and Object
  - Types mutate
- Operators
  - You use operators to manipulate type

22