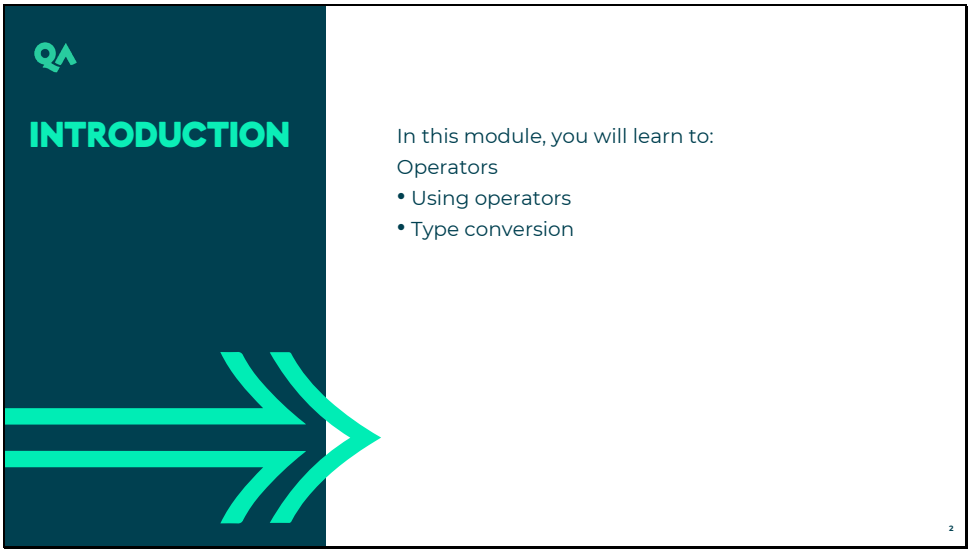
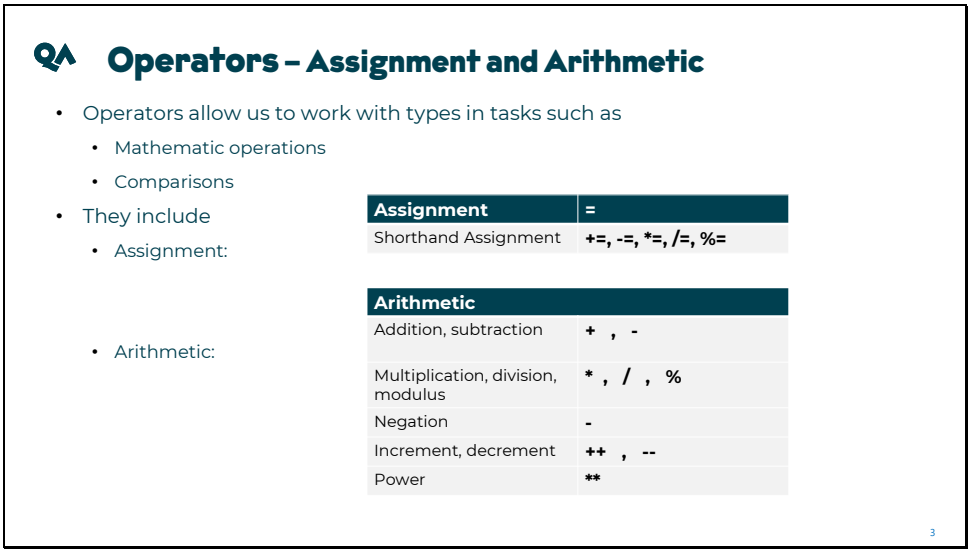
Slide 1

Slide 2


INTRODUCTION
In this module, you will learn to:
Operators
• Using operators
• Type conversion
2

Slide 3



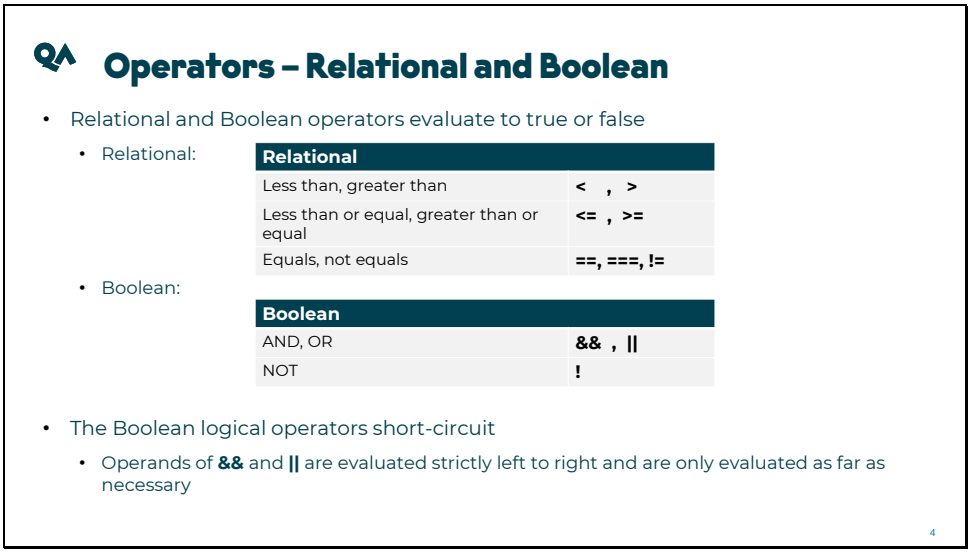
JavaScript has all the operators that you would expect for a modern language; in general, they follow the same representation as first became popular in the C language.

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:

```
let 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
```

Slide 4

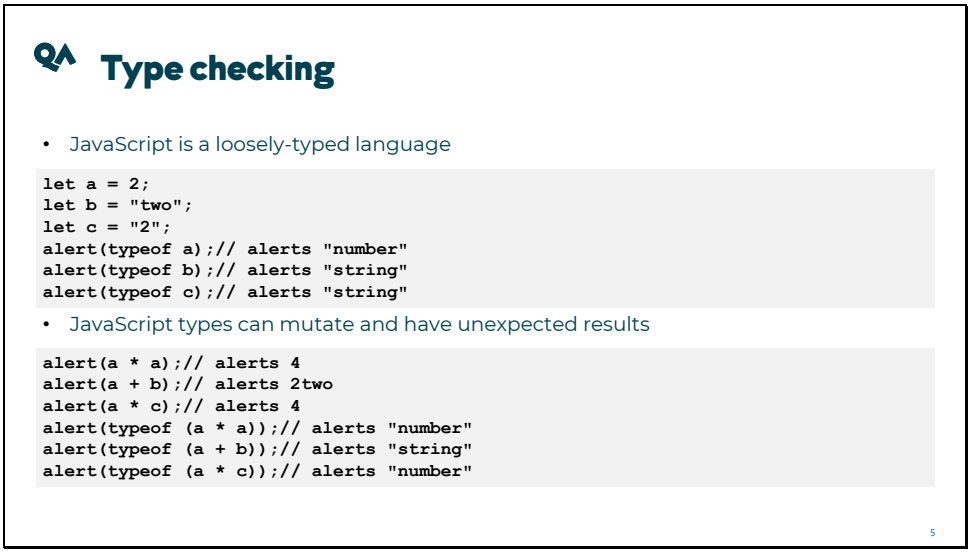


The Boolean AND and OR operators have 'short-circuit' evaluation. This means that when an expression involving them is evaluated, it is only evaluated as far as is necessary. For example, consider the expression:

```
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.

Slide 5

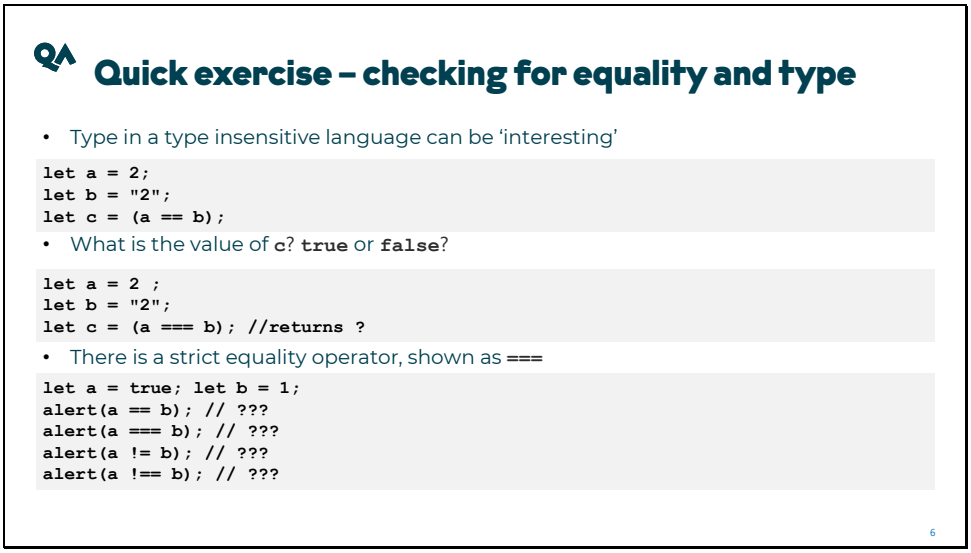


When we 'add' a string and a number using the + operator, JavaScript assumes we're trying to concatenate the two, so it creates a new string. It would appear to change the number's variable type to string. When we use the multiplication operator (*) though, JavaScript assumes that we want to treat the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

Slide 6



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.

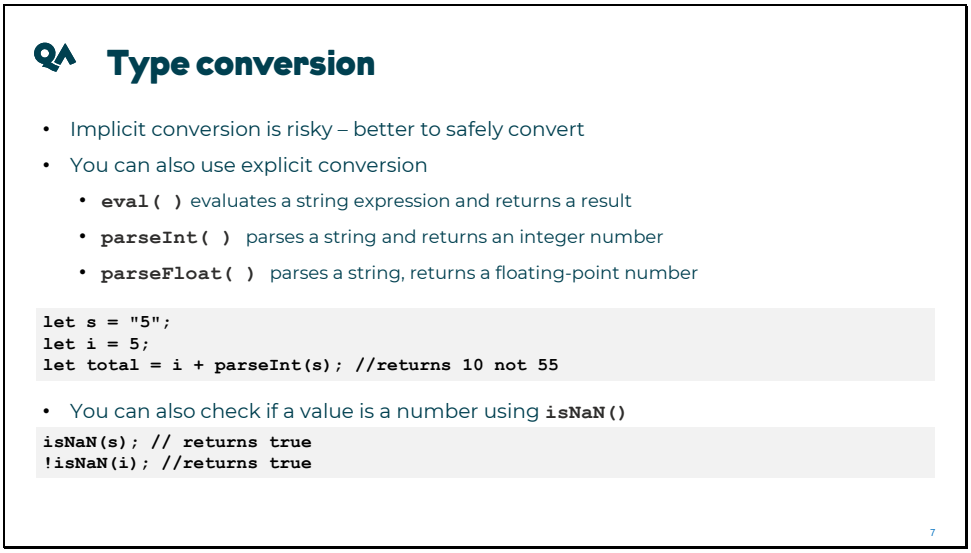
In JavaScript 1.3, the situation became even less simple, with the introduction of one further operator: the strict equality operator, shown as ===.

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:

```
let 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.

Slide 7



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:

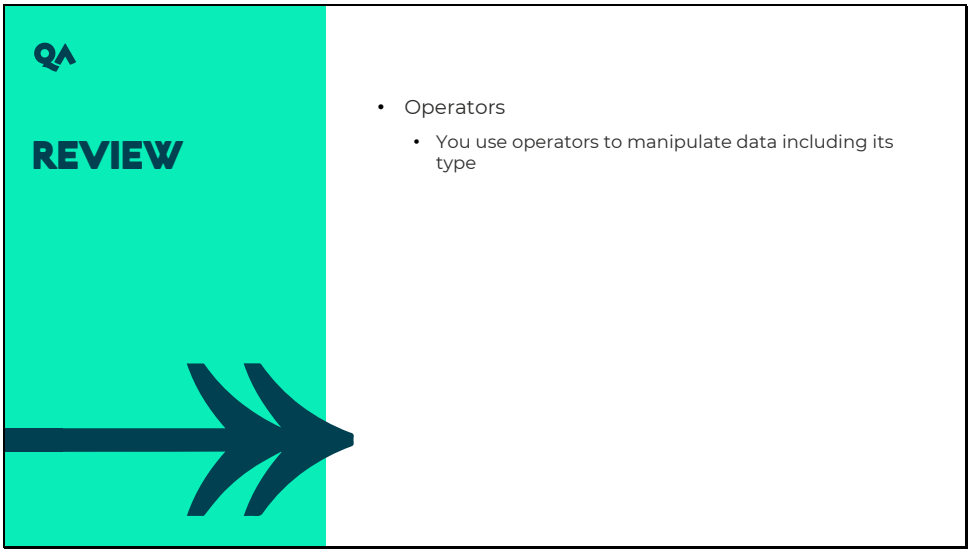
> *parseInt(55.95)* would return 55, note that no rounding has occurred
> *parseInt("55.95boom!")* would also return 55

`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.

Slide 8


REVIEW
• Operators
• You use operators to manipulate data including its type

Slide 9

QuickLab 3 - Operators

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