


INTRODUCTION

In this module, you will learn to:

- Operators
- Using operators
- Type conversion



2

QA Operators – Assignment and Arithmetic

- Operators allow us to work with types in tasks such as
 - Mathematic operations
 - Comparisons
- They include
 - Assignment:

Assignment	=
Shorthand Assignment	<code>+=, -=, *=, /=, %=</code>

- Arithmetic:

Arithmetic	
Addition, subtraction	<code>+, -</code>
Multiplication, division, modulus	<code>*, /, %</code>
Negation	<code>-</code>
Increment, decrement	<code>++, --</code>
Power	<code>**</code>


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
```



Operators – Relational and Boolean

- Relational and Boolean operators evaluate to true or false
 - Relational:

Relational	
Less than, greater than	< , >
Less than or equal, greater than or equal	<= , >=
Equals, not equals	==, ==, !=
 - Boolean:

Boolean	
AND, OR	&& ,
NOT	!
- The Boolean logical operators short-circuit
 - Operands of **&&** and **||** are evaluated strictly left to right and are only evaluated as far as necessary

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.

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"
```

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
```



Quick exercise – checking for equality and type

- Type in a type insensitive language can be 'interesting'

```
let a = 2;
let b = "2";
let c = (a == b);
```

- What is the value of **c**? **true** or **false**?

```
let a = 2 ;
let b = "2";
let c = (a === b); //returns ?
```

- There is a strict equality operator, shown as **===**

```
let a = true; let b = 1;
alert(a == b); // ???
alert(a === b); // ???
alert(a != b); // ???
alert(a !== b); // ???
```

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.

Type conversion

- Implicit conversion is risky – better to safely convert
- You can also use explicit conversion
 - `eval()` evaluates a string expression and returns a result
 - `parseInt()` parses a string and returns an integer number
 - `parseFloat()` parses a string, returns a floating-point number

```
let s = "5";  
let i = 5;  
let total = i + parseInt(s); //returns 10 not 55
```

- You can also check if a value is a number using `isNaN()`

```
isNaN(s); // returns true  
!isNaN(i); //returns true
```

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.



REVIEW



- Operators
 - You use operators to manipulate data including its type



QuickLab 3 - Operators

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