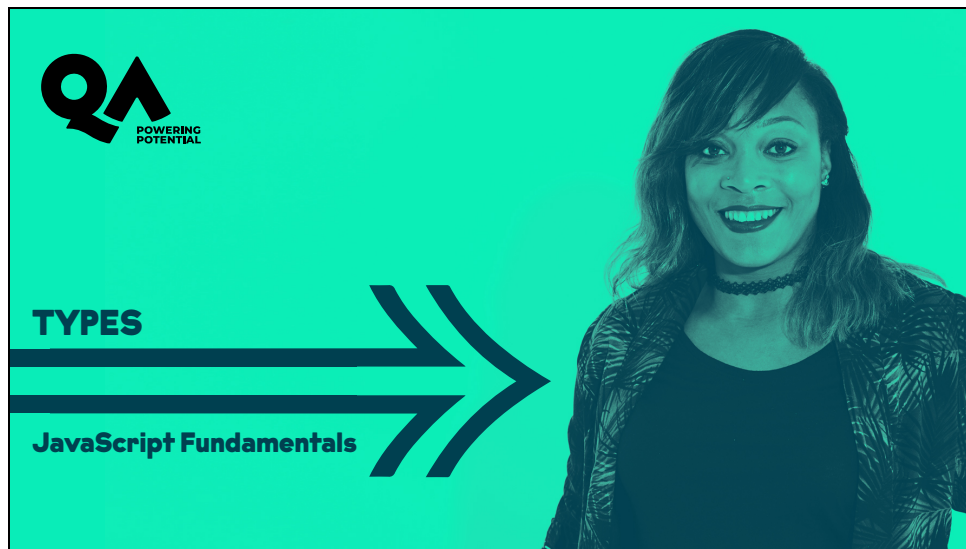




Slide 1





## INTRODUCTION



**In this module, you will learn to:**

- **Declare variables**
- **Understand types**
  - Primitive types
  - Strings
  - Numbers
  - Booleans
  - Undefined
  - Nulls
  - Symbol
  - Reference types

2



## Declaring variables

- Declaring variables
  - **const**, **let** and **var**
  - With and without assignment
  - Do not use implicit declaration
- **let** – a block-scoped variable (don't worry – we'll discuss what block-scoped means later)
- **const** – the same as **let**, but must be initialised at declaration and cannot be changed
- **var** – a function-scoped variable whose declaration is hoisted and can lead to confusing code! To be avoided now that we have **let** and **const**
- What should you use? **const** where possible. **let** when you need it to change

### Variable Declarations

```
x = 10;    // implicit - DO NOT USE
let y;     // explicit without assignment
let y = 15; // explicit with assignment
const z = 10; // constant with assignment
```

Variables in JavaScript should be declared with the **let** or **const** 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.



## Declaring variables


- Variable names
  - Start with a letter, "\_" or "\$"
  - May also include digits
  - Are case sensitive
  - Cannot use reserved keywords
  - E.g. int, else, case
- Best practice is to use camelCase for variable names

4

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.



## JavaScript types

- Dynamically typed
  - Data types not declared and not known until runtime
  - Variable types can mutate
- Interpreted
  - Stored as text
  - Interpreted into machine instructions and stored in memory as the program runs

- Primitive data types
  - Boolean
  - Number
  - String
  - Undefined
  - Null
  - Symbol
- Object

5

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.



## Primitives and Object types

- JavaScript can hold two types:
- **Primitives**
  - Primitive values are immutable pieces of data
  - Their value is stored in the location the variable accesses
  - They have a fixed length
  - Quick to look up
- **Object**
  - Objects are collections of properties
  - The value stored in the variable is a reference to the object in memory
  - Objects are mutable

6

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.



## The typeof operator

- The **typeof** operator takes on parameter the value to check

### The typeof operator

```
const TYPE_TEST = "string value";  
alert(typeof TYPE_TEST) //outputs "string"  
alert(typeof 95) //outputs "number"
```

- Calling **typeof** on a variable or value returns one of the following:
  - number
  - boolean
  - string
  - undefined
  - symbol
  - object (if a null or a reference type)

`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 not undefined

- **null** and **undefined** are different concepts in JavaScript
  - **undefined** variables have never been initialised
  - **null** is an explicit keyword that tells the runtime it is 'empty'

```
let userID = null;  
console.log(userID); //returns null
```

- There is a foobar to be aware of with **null**:
  - **undefined** is the value of an uninitialised variable
  - **null** is a value we can assign to represent objects that don't exist

```
let userID = null;  
console.log(userID == undefined); //returns true
```

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:

```
let loggedIn = false;  
console.log(loggedIn); //returns false
```

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



## The Number type


- Always stored as 64-bit values
- If bitwise operations are performed, the 64-bit value is rounded to a 32-bit value first
- There are a number of special values

Constant	Definition
<b>Number.NaN</b> or <b>NaN</b>	Not a number
<b>Number.Infinity</b> or <b>Infinity</b>	Greatest possible value (but no numeric value)
<b>Number.POSITIVE_INFINITY</b>	Positive infinity
<b>Number.NEGATIVE_INFINITY</b>	Negative infinity
<b>Number.MAX_VALUE</b>	Largest possible number represented in the 64-bits
<b>Number.MIN_VALUE</b>	Smallest possible number represented in the 64-bits

11

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:



## 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
- As a rule, always use backticks (`)

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
```



## String functions

- The String type has string manipulation methods, including:

Method	Description
<b>indexOf()</b>	Returns the first occurrence of a character in a string
<b>charAt()</b>	Returns the character at the specified index
<b>toUpperCase()</b>	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

Further methods:




## QuickLab 2

- 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



# REVIEW



## Primitive variables

- Value types

## Understand types

- There are six primitive types and object
- **Types can mutate**