# Part 2

## DATA TYPES

JavaScript only has a small set of data types:

* Numbers
* Strings
* Booleans
* Objects
* Null
* Undefined
* Only one number type – 64bit, double
  + It’s not very accurate, so always times your data by say 100, do the math, then revert it to decimals, 0.1 + 0.1 doesn’t always equal 0.2
* NAN stands for Not a number
  + Toxic, any math performed with it will also output it.
  + NAN isn’t equal to anything, even itself, NAN = NAN is false.
  + NAN is not greater or less than NAN.
  + Even though it literally stands for Not A Number, it’s type is actually a number.
  + Number(value) produces NAN if it has a problem.
  + **parseInt(value,1)** – Second number is radix, always use it or you can get weird results.
* Null = no value
* Undefined = default value for variables and parameters
  + The missing value, value

## LOOSELY TYPED

1. Any type can be used anywhere.

2. Any variable can be used as a parameter to any function

JS is not untyped

JavaScript has a good sense of types, however it just means every type can be used everywhere

There are types, JavaScript figures them out for you

## FUNCTIONS

* Functions are first class objects
* Functions can be passed, returned, and stored just like any variable
* Functions inherit from ‘Object’ and can store name/value pairs
* Functions can appear anywhere that an expression can appear
* What JS calls a function, other languages call Lambda
* Since JS is loosely typed, you don’t have to declare parameter types in the signature
* Functions share the same namespace as variables
* Functions can be defined inside functions
* An inner function has access to variables and parameters of functions it’s within
  + This is known as Static/Lexical scoping

function myFunction(p1, p2) {

return p1 \* p2;

}

const x = myFunction(4, 3); //12

## Arguments

When a function is invoked, in addition to its parameters it gets a special parameter called **arguments**

This contains all of the arguments from the invocation

It’s similar to an array (but strictly not)

This is useful when a function takes a lot of arguments you want to process easily.

function sum(a, b, c, d, e, f, g) {

let n = arguments.length;

total = 0;

for (let i = 0; i < n; i++) {

total += arguments[i];

}

return total;

}

If you were to pass through more arguments then the 7 defined here, the arguments would exist and be used but you would have no way of accessing them directly.

## Functions – Best Practice

Whenever you’re creating a function, follow this layout:

1. **Variables go first**
2. **Then functions**
   1. **Then variables**
   2. **Then functions**
3. **Run code**

let x = 10;

function print(input) {

let x = 0;

function log() {

} console.log(input);

}

print(10);

## Return

Every function has a return type, if you don’t declare it, it’ll simply return nothing

Return stops execution of the function and returns the specified value

function  myFunction() {  
 return  Math.PI;  
}  
  
  
function  myFunction(name) {  
 return  "Hello"  + name;  
}

## Arrow Functions

* + Also known as “Fat Arrows”
  + Concise way of writing functions
  + Creates anonymous functions
  + Similar to lambdas in other languages
  + Does not require the return or function keywords or the use of curly brackets

function f(data) {

**return data + 1;**

**}**

//Arrow equivalent

const f = (data) => data + 1;

Can also have multiple parameters

If you require multiple lines in the method, you then require a return statement.

// Multiple parameters

const a = (data, moreData) => data + moreData;

// Or no parameters

const b = () => 'this one returns a string';

// Curly brackets, for multiple line statements

const c = (data) => {

data = data + 1;

data = data \* 3;

return data;

};

// Just returning an object? Wrap it in ( )

const d = (data, moreData) => ({ a: data, b: moreData });

// New style

let odd = evens.map(v => v + 1)

// Old style

let odd = evens.map(function (v) { return v + 1; });

// New Style

let output = (data) => console.log(data);

// Old Style

let output = function (data) { console.log(data); };

## var, let, const

Variables in JavaScript

Declaring variables looks very similar in JavaScript to other programming languages

However we use the **let** and **const** keyword – older examples may use the var keyword.

Declaring an object is done through the use of curly braces, with the information of that object being inside

var variableString = "hi";

let variableNumber = 3;

const variableObject = {

name: "Jordan"

};

window.alert(variableObject.name);

Similar to Java except we don’t have our types

## Var

The scope of a var depends on where it was declared:

Globally - When declared outside a function. This means it can be accessed anywhere in the window

Locally - When declared inside a function. This means it can only be accessed inside the function it was defined in

A var can be declared more than once

* **var is rarely be used in ES6. Use let or const!**

// Declaration

var x;

// Updating it's value

x = 'test string';

// Redeclaring is allowed with var!

var x = 12345;

## let

ES6 introduced the let keyword, the preferred was of declaring variable.

* lets are block scoped
* A let can’t be declared more than once
* A let can have it’s value updated

// Declaration

let x;

// Updating it's value

x = 'test string';

// This isn't allowed in the same scope!

// This line will error

let x = 12345;

## const

consts are useful for declaring constant values.

* consts are block-scoped
* A const can’t be declared more than once
* A const can’t have it’s value updated

This means when you declare a const you must initialise it with a value

While the value can’t be updated, if initialised with an object the properties of that object can be updated.

// Declaration (requires a value!)

const x = 'my const';

// Updating the value isn't allowed!

x = 'test string';

// This isn't allowed in the same scope!

// This line will error

const x = 12345;