Institute of Data

Software Engineering

Module 3 JavaScript Fundamentals

Agenda: Module 3

- JavaScript **Fundamentals**
- **Intermediate** JavaScript
- **Advanced** JavaScript

Each section has its own material and lab exercises. The material is packed with examples, designed to be copied and pasted to encourage experimentation and practical understanding as you learn each concept. The lab exercises reinforce key concepts from the material and should be done independently as you cover each section.

```
romise from setTimeout
st afterSomeTime = (time) => new Promise(resolve => {
etTimeout(() => {
 resolve(true);
}, time);
;
nst callAfterSomeTime = (callback, time) => afterSomeTime(time).then(callback);

allAfterSomeTime(() => console.log('Hello after 1500ms'), 1500);

const getData = async (url) => fetch(url);

document
  .querySelector('#submit')
  .addEventListener('click', function() {
    const name = document.querySelector('#name').value;

    // send to backend
    const user = await fetch(`/users?name=${name}`);
    const posts = await fetch(`/posts?userId=${user.id}`);
    const comments = await fetch(`/comments?post=${posts[0].id}`);
    //display comments on DOM
```

JavaScript Fundamentals

- Definition of JavaScript
- JavaScript Engine
- Code Structure
- Variables
- Data Types
- Type conversion
- Comparison
- Functions
- Objects

Definition of JavaScript

- JavaScript was initially called ‘LiveScript’ which means ‘make web pages alive’
- JavaScript is a fully independent language with its own specification called ECMAScript. e.g. ES5 (2009), ES6 (2015), ES7
- Comparing ‘JavaScript’ and ‘Java’ is like comparing ‘cart’ and ‘carpet’. They are both programming languages, and have similar names, but have almost nothing else in common.

JavaScript is the only browser technology that combines all of the above.

Why JavaScript?

- Full integration with HTML/CSS
- Simple things are done simply
- Support by all major browsers and enabled by default
- Can be used for both front-end and back-end applications

JavaScript Engine

JavaScript can execute in any environment (browser, server etc), where there is a JavaScript engine. The browser has an embedded engine called the 'JavaScript virtual machine' with different 'code names', e.g.

- V8 – in Chrome and Opera, also Edge
- SpiderMonkey – in Firefox
- JavaScriptCore, Nitro/SquirrelFish – in Safari

The primary task of the engine is to convert (compile) the JavaScript to machine-executable binaries.

On the server side we usually use Node.js to compile and execute JavaScript.

Code Structure

- JavaScript commands and constructs are written in statements, which are separated by semicolons.

```
console.log('hello'); console.log('world');
```

- Semicolons can be omitted when a line break exists.

```
console.log('hello')
console.log('world')
```

- Comments are used for text not compiled by the JS engine

```
console.log('world') // this comment follows the statement, will be ignored when compiled
```

Code Structure

- Line comments tell the JS engine to ignore anything after the **//** on a single line
- Block comments begin with **/*** and end with ***/** - everything in between is ignored as a comment even if it spans multiple lines

```
/* code block comment
    function printConsoleMessage() {
        console.log('I am commented out')
    }
*/
```

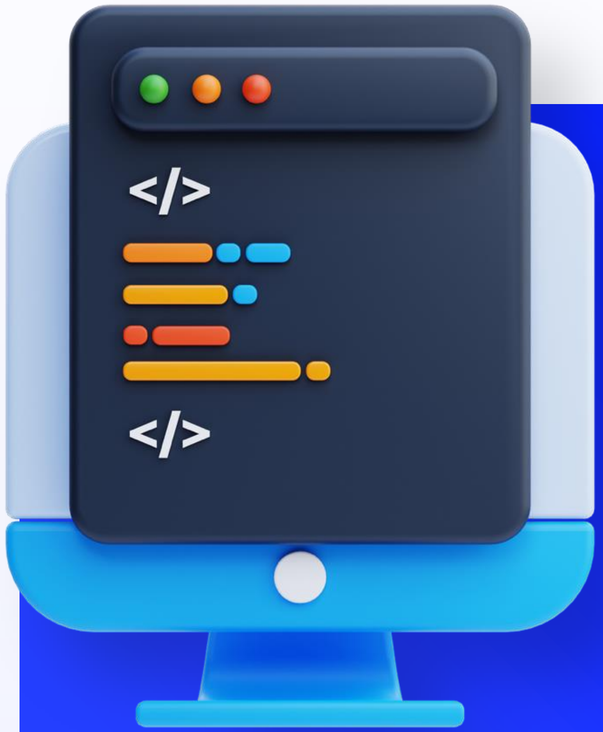
- *Tip: in VS Code, use the shortcut **Ctrl-K-C** to make a comment, and **Ctrl-K-U** to uncomment. Try it!*

Variables

A variable is a 'named storage' for data. **const**, **var**, **let** are keywords used to declare (or create) a new variable.

Variable naming

- Names can contain only **letters**, **digits**, or **symbols $** and _
- The first character **cannot be a digit**
- Reserved names cannot be used, e.g. **let**, **class**, **return**, **function**
- The name is **case-sensitive**
- **Non-Latin** letters are allowed, but not recommended



Data Types

Number

- The number type represents both integer and floating point numbers.

```
const integer = 123 // integer - whole number
const float = 12.345 // floating point - decimal number
```

- Special numeric values: **Infinity**, **-Infinity**, **NaN** (Not a Number)

```
console.log(1/0) // division by zero = Infinity
console.log(-1/0) // negative division by zero = -Infinity
console.log("not a number" / 1) // invalid mathematical operation = NaN
```

Data Types

Mathematics with numbers

- +, -, / and * are used to calculate maths results from numeric variables

```
let one = 1;
let two = 2;
let three = 3;

// standard BODMAS order of operations - use brackets to
override
// below is: 1 + 2 - ( (3 * 2) / 1 ) = 3 - 6
console.log(one + two - three * two / one); // -3
```

- Shortcuts like += and ++ can modify a number using mathematical operations:

```
// to increment by one (all the same):
one = one + 1; // new value of one is previous value + 1
one += 1; // shorthand - add 1 to previous (also *=)
one++; // increment previous value (by 1)

// to decrement by one (all the same):
two = two - 1; // new value of two is previous value - 1
two -= 1; // shorthand - minus 1 from previous (also /=)
two--; // decrement previous value (by 1)
```

Data Types

BigInt

In JavaScript, the number type cannot represent integer values larger than (2**53-1) (that’s 9,007,199,254,740,991 - over 9 quadrillion), or less than -(2**53-1) for negatives. It’s a **technical limitation** caused by their internal representation, which doesn’t often cause issues.

BigInt value for numbers outside of this range is created by appending **n** to the end of an integer, which allows it additional memory space.

```
const bigint_valid = 1234567890123456789012345n;
const bigint_invalid = 1234567890123456789012345; // too large for standard integers

console.log(bigint_valid == bigint_invalid) // false
```

Institute of Data

Data Types

String

A string in JavaScript must be surrounded by quotes. There are three types of quotes, all of which are fundamentally the same.

- **Double quotes** - useful when the string contains an apostrophe

```
const doubleQuotes = "String that can include 'single quotes'"
```

- **Single quotes** - useful when the string contains double quotes

```
const singleQuotes = 'String that can include "double quotes"'
```

- **Backticks** - useful when embedding expressions/variables

```
const backTicks = `String that can include variables - ${singleQuotes}`
```

Data Types

Boolean

The Boolean type has only two values: **true** and **false**.

This type is commonly used to store **yes/no** or **on/off** values.

```
let isChecked = false;
let isToggleOn = true;
```

We can use the **not** operator **!** to switch between these two possible values:

```
isChecked = !isChecked;
console.log(isChecked); // true
```

Data Types

Null

In JavaScript, null is not a **reference to a non-existing object** nor a **null pointer** like in some other languages.

It's simply a special value which represents **nothing**, **empty** or **value unknown**.

```
let age = null
```

null counts as false when converted to a boolean, and 0 when converted to a number value. It usually indicates that while a variable has an intentionally empty value at this point in time, it may be assigned a value later.

Data Types

Undefined

The special value **undefined** makes a type of its own, similar to **null** but slightly different.

The meaning of undefined is that the **variable exists** but its **value is not assigned**.

If a value is **declared**, but **not assigned**, then its value is **undefined**.

```
let location; // no value is assigned with the = operator
console.log(location); // undefined
```

Null is equivalent to **undefined**, but they are not strictly equal:

```
let location; // no assigned value, therefore undefined
let age = null; // explicitly assigned null value

console.log(`${location} == ${age} ? ${location == age}`) // true (uses == to check value equivalence)
console.log(`${location} === ${age} ? ${location === age}`) // false (uses === to check type equality)
```

Data Types

Object

The object type is special. All other types are called “**primitive**” because they represent only a **single value** (this could be a string, number, boolean, etc). In contrast, **objects** are used to store **collections of data**, usually using **key-value pairs** where the keys (names) are strings. **Arrays** are a special type of object containing **numeric keys** (indexes) with a specific length.

Symbol

The symbol type is used to create **unique identifiers for objects** and in general development is used **very little**.

Data Types

Object

Objects are typically used to represent an entity and store all related data, eg. for a customer, a product, an order, etc. Curly brackets are used to indicate where the object begins and ends. Inside, a series of key (property name) and value pairs are separated by commas, with colons separating the **key** and **value**.

```
const tv = { // object starts here
  brand: "Sony Bravia", // key-value pair. brand is the key, "Sony Bravia" is the value
  size: "55-inch", // values can be any data type
  model: 2023, // multiple key-value pairs are separated by commas
  resolution: "4K" // the comma on the last key-value pair can be omitted
} // object ends here. All data is stored in tv variable.
```

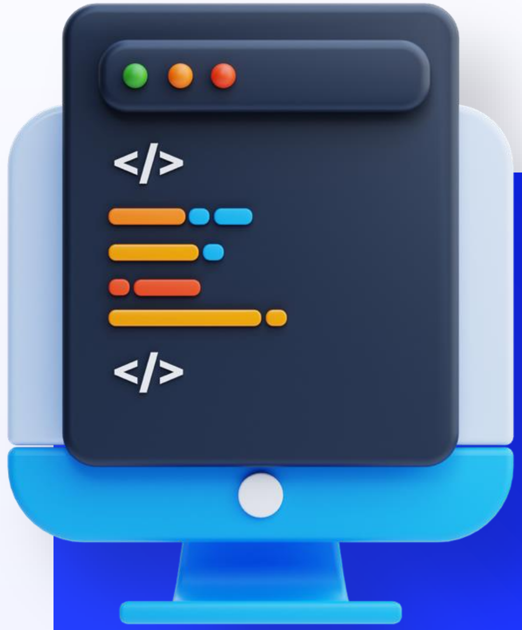
Data Types

The typeof operator

The **typeof** operator returns the **type** of the argument. It's used when we want to process values of different types differently or want to do a quick check.

It supports two forms of syntax:

- As an **operator** (most common): **typeof x**
- As a **function**: **typeof(x)**



Data Types

Type check examples:

```
console.log(typeof undefined); // undefined
console.log(typeof 0); // number
console.log(typeof 10n); // bigint
console.log(typeof true); // boolean
console.log(typeof 'text'); // string
console.log(typeof Symbol('id')) // symbol
console.log(typeof Math); // object
console.log(typeof null); // object
console.log(typeof console.log); // function
```

Note: the result of **typeof null** is **"object"**. That's an **officially recognised error** in **typeof** behaviour. Try some more of your own!

Institute of Data

Data Types

JavaScript is a **dynamically typed** language. This means you don't have to specify the data type of a variable when you create it. It also means that data types can change, and are automatically converted as-needed during script execution.

The valid primitive JS data types are: **null**, **undefined**, **Boolean**, **Number**, **BigInt**, **String** and **Symbol**. **Objects** are an additional complex data type, with **Arrays** as a special kind of object used for collections, and **Functions** as a special kind of object used for grouping and executing statements. We will cover each in turn.

We can **explicitly** convert values from one type to another, or rely on some built-in ways that JavaScript will **implicitly** convert types for us.

Type Conversion

String conversion

Converting other types of variables to **strings** happens when we need the string form of a value. We can do this **explicitly** or **implicitly**:

- **Explicit** conversion - by using **String** class function

```
console.log( String(false) ) // false - string form of boolean
```

- **Implicit** conversion - by **concatenating** an existing string

```
console.log( "1" + 2 + 3 ) // 123 - string ‘1’ is concatenated with number 2 then number 3
console.log( 1 + 2 + "3" ) // 33 - number 1 is added to number 2 then concatenated with string ‘3’
```

Type Conversion

Numeric conversion

Conversion of other types of variables to **numbers** happens in **mathematical** functions and expressions **automatically**, or we can do explicit **conversions**.

- **Explicit** conversion - by using **Number** class function

```
console.log( Number("      4      ") ) // 4 - trims spaces
console.log( Number(null) ) // 0 - intentionally empty value converts to 0
console.log( Number(undefined) ) // NaN - non-existent value is unknown
console.log( Number(false) ) // 0 - false converts to 0
console.log( Number(true) ) // 1 - true converts to 1
console.log( Number("") ) // 0 - empty string converts to 0
console.log( Number("not a number") ) // NaN - non-empty strings beginning with chars cannot convert
```

Type Conversion

Numeric conversion

Implicit conversion happens when using division **/**, multiplication *****, subtraction **-**, or unary plus **+**, and JS **automatically converts strings to numbers** (where possible) in order to perform the operation.

```
console.log("6" / "2") // 3
console.log("6" * "2") // 12
console.log("6" - "2") // 4
console.log(+"6") // 6
```

Note: implicit numeric conversion **does NOT happen** when using **+** with two or more operands, as JS can (and will) **'add'** two strings by **concatenating** them. String concatenation is the operation of joining character strings end-to-end.

Type Conversion

Boolean conversion

Boolean values are used in **logical conditions**. We can explicitly **convert/create** them by using the **class function**:

```
console.log( Boolean("") ) // false - empty string
console.log( Boolean(0) ) // false - zero value
console.log( Boolean(null) ) // false - no value
console.log( Boolean(undefined) ) // false - unknown value
console.log( Boolean(NaN) ) // false - not a number
console.log( Boolean("false") ) // true - non-empty string
console.log( Boolean(-1) ) // true - non-zero number
```

Type Conversion

Boolean conversion

Implicit boolean conversions via an **if condition**, **conditional statement** or **negation** are far more common:

```
if ("") console.log('empty string is true') // implicit "" conversion to false - won't print msg
if (undefined) console.log('undefined is true') // implicit conversion to false - won't print msg
```

Conditional statement:

```
console.log( NaN ? 'NaN is true' : 'NaN is false' ) // NaN is false
console.log( 0 ? 'zero is true' : 'zero is false' ) // zero is false
console.log( "hello" ? 'hello is true' : 'hello is false' ) // hello is true
```

Negation:

```
console.log( !undefined ) // true - convert value to boolean then negate it (opposite)
console.log( !!"" ) // false - convert value to boolean then negate/toggle twice
```

Comparison

String comparison

To see whether one string is greater than another, JavaScript uses **dictionary** or **lexicographical** order. In other words, strings are compared **letter-by-letter**.

The algorithm to compare two strings is simple:

1. Compare the first character of both strings.
2. If the first character from the first string is greater (or less) than the other string's, then the first string is greater (or less) than the second. We're done.
3. Otherwise, if both strings' first characters are the same, compare the second characters the same way.
4. Repeat until the end of either string.
5. If both strings end at the same length, then they are equal. Otherwise, the longer string is greater.

```
console.log('apple' < 'banana'); // true - because a is less than b (rule 2)
console.log('eat' < 'eaten'); // true - because all characters are the same but eaten is longer (rule 5)
```

Comparison

Comparison of different types

When comparing values of different types using **>**, **<, ==** and **!=**, JavaScript **converts the values to numbers**.

```
console.log("2" > 1) // true - converts to 2 > 1
console.log("2" != 1) // true - converts to 2 is not equal to 1
console.log("02" == 2) // true - converts to 2 == 2
console.log(true == 1) // true - true converts to 1
console.log(false == 0) // true - false converts to 0
console.log(null == undefined) // true - both convert to 0
```

Replacing the **==** (equivalence) in the above with **===** (strict equality) will check for **matching types** as well and change the results to **false**. Try it!

Functions

Definition

Functions are the main **building blocks** of a program. They allow blocks of code to be **defined once** and called many times, avoiding the need to repeat ourselves. DRY (**Don't Repeat Yourself**) is a common coding maxim for writing succinct and efficient code, and good use of functions is a core part of this.

Function declaration syntax

The **function** keyword goes first, then goes the **name** of the function, then a list of **parameters** between the parentheses (**comma-separated**) and finally the **code** of the function, also named the **function body**, between **curly braces**.

Functions - Declaration

```
// 'function' keyword followed by the custom function name, then ()
function helloWorld() // we can include parameters inside the () as function variables
{ // function body is enclosed with curly brackets (braces)
    console.log('hello world'); // can be one or multiple statements inside the function
}

helloWorld(); // once the function is defined, we need to call it to make it run/execute
```

The above function is declared/called **‘helloWorld’** and just prints a simple message. It **won’t run until we call it**, which we can do as many times as needed.

This function will print ‘hello world’ to the console and then end, without **‘returning’** any value or result.

Functions - Declaration

Functions can also **‘return’** a result when called. The **return** keyword is the **last thing** that happens in a function and will **send a value** back to where it was called. We can then log this value (as below) or store it in a variable.

```
// function checkAge returns a value when called
function checkAge(age) {
    if (age >= 18) {
        return 'adult'; // if the condition is true, return this string and exit
    }
    return 'non-adult'; // if the condition was false, return this string and exit
}

console.log( checkAge(21) ) // adult
console.log( checkAge(13) ) // non-adult
```

Functions - Declaration

Functions can also be created as a **variable expression** using the syntax below. In this case, the function is **assigned to the variable** explicitly, like any other value.

No matter how the function is defined, it’s still a value stored in a named variable. Return statements **work the same** for both function declarations and function expressions.

```
const sayHi = function() { // function expression syntax for creating a function
    console.log('Hi')
}
```

Functions

Function expression vs Function declaration

- Function declaration can be **hoisted** (called earlier than it is defined), whereas function expression can only be accessed ***after*** definition

```
sayHiExpression(); // error - cannot access before initialization
sayHiDeclaration(); // works - can be hoisted

const sayHiExpression = function() {
    console.log('Hi');
}

function sayHiDeclaration() {
    console.log('Hi');
}
```

Functions - Arrow

Function expression vs Function declaration

Arrow functions are a more concise, **shorter version** of function expression. We still use the variable expression syntax, but can omit the function keyword. If the **function** contains only a single statement in the body, we can also omit the **braces**.

We do use an arrow **=>** prior to the body of the function.

```
const sayHiArrow = () => console.log('Hi') // arrow function syntax, more concise
```

The below arrow functions are equivalent. Both accept two parameters and return the difference. We can omit the **braces** and **return** keyword.

```
const subtract1 = (a, b) => a - b; // most concise version of the below
const subtract2 = (a, b) => { return a - b }; // does the same thing as above
```

Functions - Arrow

Function expressions and declarations can access an **arguments** variable within the body, but arrow functions cannot:

There are other more **technical differences** as well. Arrow functions are most useful when **passed as parameters** to other functions. We will cover more examples in later material.

```
const sayHiExpression = function() {
    console.log('Hi');
    console.log(arguments);
}

function sayHiDeclaration() {
    console.log('Hi');
    console.log(arguments);
}

const sayHiArrow = () => {
    console.log('Hi');
    console.log(arguments);
}

sayHiDeclaration() // [Arguments] {}
sayHiExpression() // [Arguments] {}
sayHiArrow() // ReferenceError: arguments is not defined
```

Objects

Definition

An object is a **complex variable** which stores an **optional list of properties**, used to store **keyed collections** of various data and more complex entities.

A property is a **key:value** pair, where **key** is a string (also called a **property name**), and value can be **anything**.

Creation

An object can be created with figure/curly brackets **{…}** with an **optional list of properties**.

Type Conversion

Empty Object

An **empty object** ('empty cabinet') can be created using one of two syntaxes:

- Object constructor

```
const user = new Object(); // creates a new empty object
```

- Object literal (more common)

```
const user = {}; // creates a new empty object
```

Objects

Object with properties

- We can immediately put some properties into **{…}** as **key:value** pairs. A property has a **key** (also known as **name** or **identifier**) before the colon **:** and a **value** to the right of it.

```
const user = { // user object contained within curly braces
    name: 'joe', // string property with key 'name' and value 'joe'
    age: 20, // numeric property with key 'age' and value 20
    'has a dog': true // multi-word property key 'has a dog' with boolean value true
};
```

Objects

Object with properties

- We can **get**, **set** or **delete** the value of a property key.

```
console.log(user.name); // get object property called name and log it
let dogOwner = user['has a dog']; // get value of property 'has a dog' and assign to new variable
user.age = 21; // set (or assign) new value to object property called age

user.location = 'NSW'; // create new object property called location and set (assign) a value
delete user.location; // delete property of user object called location
```

Try it: Follow the example to create an object representing yourself, then get, set and delete some of the properties.

Objects

Property name limitation

- Object property names (keys) are **automatically converted to strings**. So be careful if using both numeric and string property names/keys:

```
const object = {
    2: 'value of numeric property',
    '2': 'value of string property'
}

console.log(object) // { '2': 'value of string property' } since 2 and ‘2’ are same
```

Objects

Property existence test

It is possible to access any property of an object. Reading a non-existing property just returns undefined. So we can take advantage of **implicit boolean conversion** to check whether the property exists.

```
const phone = {
    model: 'iPhone 11',
    colour: 'black'
}

if (phone.colour) console.log(`My ${phone.model} is ${phone.colour}`) // prints message
if (phone.storage) { // undefined counts as false, so the below won't print
    console.log(`My ${phone.model} has ${phone.storage}GB`);
}
```

Objects

Iteration

When we have **multiple** things stored in a single variable, it is common to want to **loop over** each one of them and perform the same operation. There are **many** options for achieving this in JS depending on the **type of data** and operation. One of the most basic is a **for loop**:

```
let goal = 5;
for (let i = 0; i < goal; i++) {
    console.log(`Iteration ${i} of ${goal}`)
}
```

We typically have an **index variable** (**i**) to keep track of which iteration we are up to, a **limit or total** number of iterations, and a way to move to the **next index**.

Objects

Object Iteration

To walk over all keys of an object, there exists a special form of for loop: for … in.

```
const phone = {
    model: 'iPhone 11',
    colour: 'black',
    storage: 64
}

for (let key in phone) { // iterates over each property in the phone object, stores in ‘key’ variable
    console.log('key: ' + key); // prints each object property name (key) in turn
    console.log('value: ' + phone[key]); // prints each object value in turn
}
```

Try it: use a for … in loop to iterate over the object representing yourself!

Objects

References

Objects are stored and copied **by reference**, whereas primitive values: **strings**, **numbers**, **booleans**, etc., are always copied **as a whole value**. A variable assigned to an object stores not the object itself, but its **address in memory**, in other words ***a reference*** to it.

```javascript
let person1 = { name: 'Anna' }; // object - stored by reference
let person2 = person1; // person2 points to same memory location as person1
person1.name = 'Brian';
console.log(person2.name); // Brian, even though we changed person1.name
let person3 = 'Carly'; // string - stored by value
let person4 = person3; // person4 points to separate memory location than person3, but both store same value
person3 = 'David';
console.log(person4); // still Carly, since person3 and person4 are string primitives and store independent values
```

Objects

Shallow Copy

Creates a new object and replicates the structure of the existing one by **iterating over its properties** and copying them on the **primitive level**.

```
const user = { name: 'Elliot', age: 27 };

const userClone = {}; // empty object, refers to different memory location

for (let key in user) { // iterate over user properties
    userClone[key] = user[key]; // re-create them in userClone
}

console.log(userClone); // { name: 'Elliot', age: 27 }
```

Objects

Shallow Copy with Object spread

Simpler and more common to create a ‘shallow copy’ with spread syntax ...

```
const userClone = {...user}; // spread or unpack user properties into new object
console.log(userClone); // { name: 'Elliot', age: 27 }
```

We can also **override values** or add in **new properties** while doing this:

```
const userClone = {...user, age: 28, location: 'New Zealand'};
console.log(userClone); // { name: 'Elliot', age: 28, location: 'New Zealand' }
```

Or merge **multiple** objects into a **single** new object:

```
const vehicle = { make: 'Toyota', model: 'Camry'};
const mergedUser = {...user, ...vehicle};
console.log(mergedUser); // { name: 'Elliot', age: 27, make: 'Toyota', model: 'Camry' }
```

Objects

Deep clone

Shallow cloning will still result in **references** to **non-primitive** object properties (such as nested objects). Deep cloning (not natively supported, see Lodash Documentation and Deep copy will clone **all levels** of an object.

```
const box1 = {
    weight: '20kg',
    dimensions: { // nested object property
        width: '30cm',
        height: '10cm'
    }
}
const box2 = {...box1}; // shallow clone
box1.dimensions.height = '12cm'; // change box1 nested object property
console.log(box2); // box2 references box1 dimensions and picks up height change
```

Objects

Methods

A **function** that is a property of an object is called its **method**.

```
const user = {
    name: 'Bilbo Baggins',
    sing: function() { // method of user object
        console.log('Roads go ever ever on');
    },
    sing2() { // shorthand method syntax, does same as above
        console.log('Over rock and under tree');
    }
}
user.sing(); // Roads go ever ever on
user.sing2(); // Over rock and under tree
```

Objects

Methods with context

It is common that an object method needs to **access** the information stored in the object to do its job. To access the object, methods can use the this keyword. The value of this is the object **before dot**, the one used to call the method.

```
const user = {
    name: 'Bilbo Baggins',
    printGreeting() {
        console.log(`Hello, I'm ${this.name}`) // 'this' is the current object
    }
}
// 'user' is before the dot, provides the context where 'this' comes from
user.printGreeting(); // Hello, I'm Bilbo Baggins
```

Objects

this behaviour

The value of this is evaluated during run-time, depending on the context. We will revisit ways to manage it, but for now consider it a **generic way to reference the current object**.

```
const user = {
    name: 'Bilbo Baggins',
    printThis() {
        console.log(this); // 'this' is the current object
        return this; // if we return it, we can 'chain' object methods together
    },
    printGreeting() {
        console.log(`Hello, I'm ${this.name}`); // 'this' is the current object
    }
}
user.printThis().printGreeting(); // methods chained together, works because printThis returns this
```

Objects

Constructor function

Standard object {...} syntax allows us to create **one object**. But often we need to create **many similar objects**, like multiple users or menu items and so on. That can be done in a few ways, such as using **constructor functions** and the new operator.

- Constructor functions are named with a **capital letter first**.
- Constructor functions should be executed only with “**new**” operators.
- Constructor functions can contain other functions

```
function User(first, last) { // constructor function
    this.first = first;
    this.last = last;
    this.hasShortName = () => this.first.length <= 3;
}

// we can create multiple users with different names
let user1 = new User('Tim', 'Smith'); // need to use 'new'
console.log(user1); // User { first: 'Tim', last: 'Smith' }
console.log(user1.hasShortName()); // true
```

Institute of Data

Objects

Object generations with new

When a function is executed with new, it does the following steps:

- A new empty object is created and assigned to this.
- The function body executes. Usually it modifies this, adds new properties to it.
- The value of this is returned.

```
function User(first, last) { // constructor function
    //this = {}; // implicitly
    this.first = first;
    this.last = last;
    this.hasShortName = () => this.first.length <= 3;
    //return this; // implicitly
}
```

Objects

Object generation in ES6

In ES6, the new syntax keyword class is introduced. Nowadays we mostly use classes instead of constructor functions to **set up blueprints for objects** and create **multiple instances**.

Classes have an explicit constructor function and a slightly different **internal syntax**, but still use the **new** syntax when creating. We will revisit them in later material.

```
class User {
    constructor(first, last) {
        this.first = first;
        this.last = last;
    }

    hasShortName() {
        return this.first.length >= 3
    }
}

let user2 = new User('Tina', 'Smith') // need to use 'new'
console.log(user2) // User { first: 'Tina', last: 'Smith' }
console.log(user2.hasShortName()) // false
```

Revision

- What are 3 advantages of programming in JavaScript?
- What is a variable, and what are the 3 keywords used to create one?
- What are 5 common variable data types?
- What are some different types of numbers supported in JS?
- What are the 3 types of quotes used to create a string?
- What is the difference between null and undefined?
- How can we convert a variable to a string? To a number?
- What is a boolean?
- What is a function? What are the 3 ways to create one?
- What is an object? How can we create one?