



Programming with Objects - Module 1

Noroff Guide



**Noroff
Education**

Table of Contents

Module overview	3
Introduction	3
Learning outcomes	3
1.1. Lesson - Introduction to Objects (part 1)	5
Introduction	5
Learning outcomes	6
Introduction to Objects	6
Object containers.....	8
Object properties.....	11
Accessing objects	12
Object methods.....	15
What did I learn in this lesson?	17
References	17
1.1. Lesson task - Introduction to Objects (part 1).....	17
The task.....	17
1.2. Lesson - Introduction to Objects (part 2)	18
Introduction	18
Learning outcomes	19
Classes as objects	19
Object constructors.....	22
Object prototypes.....	24
Object iterables.....	26
Global Objects	28
What did I learn in this lesson?	30
References	31
1.2. Lesson task - Introduction to Objects (part 2).....	31
The task.....	31
1.3. Lesson - JSON (part 1)	32
Introduction	32
Learning outcomes	33
Introduction to JSON.....	33
Building JSON structure	35
JSON libraries and programming tools.....	36

JSON conversion.....	38
JSON and other data formats.....	41
What did I learn in this lesson?	45
1.3. Lesson task - JSON (part 1).....	45
The task.....	45
1.4. Lesson - JSON (part 2)	46
Introduction	46
Learning outcomes	47
Applying techniques for working with JSON.....	47
Introduction to JSON Schema	50
Working with JSON Schema	52
JSON-LD	56
What did I learn in this lesson?	58
References	59
1.4. Lesson task - JSON (part 2).....	59
The task.....	59
1.5. Lesson - Self-study	60
Introduction	60
Materials	61
1.5. Lesson task - Self-study.....	61
The task.....	61

Module overview

Introduction

Welcome to Module 1 of Programming with Objects.

If anything is unclear, check your progression plan and/or contact a tutor on Discord.

Module structure	Estimated workload
1.1. Lesson and task Introduction to Objects (part 1)	8 hours
1.2. Lesson and task Introduction to Objects (part 2)	8 hours
1.3. Lesson and task JSON (part 1)	8 hours
1.4. Lesson and task 1.4 JSON (part 2)	8 hours
1.5. Lesson and task Self-study	8 hours

Learning outcomes

In this module, we are covering the following knowledge learning outcomes:

- The candidate has knowledge of concepts, processes and tools in Object Orientated programming and JavaScript Object Notation.

- The candidate has knowledge of concepts and processes of functional programming as an alternative to Object Orientated programming.
- The candidate has knowledge of concepts, processes and tools used with existing REST APIs.
- The candidate can update their REST API and advanced JavaScript language knowledge.
- The candidate has knowledge of the industry and the JavaScript Developer role with the associated responsibilities.

In this module, we are covering the following skill learning outcomes:

- The candidate can apply vocational knowledge of Object Orientation concepts to practical and theoretical problems.
- The candidate masters JavaScript Object Notation techniques and styles.
- The candidate masters advanced JavaScript tools, materials, techniques and styles.
- The candidate can find information and material relevant to using existing REST APIs.
- The candidate can find information and materials needed to build general-purpose JavaScript solutions to relevant problems.

In this module, we are covering the following general competence learning outcome:

- The candidate can carry out work based on the needs of a junior JavaScript developer.

1.1. Lesson - Introduction to Objects (part 1)

Introduction



In JavaScript, all values, besides primitives, are Objects. Primitives are types and values such as string, number, boolean, null, undefined, symbol or BigInt. However, even these can be represented as Objects.

We can treat Objects as containers for data. Object properties and Object methods can store numerous variables and functions.

In this lesson, we'll recall basic information about Objects, how to create them and how to access or modify their properties and methods. More complex details on Objects will be presented in the next lesson.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of concepts, processes and tools in Object Orientated programming and JavaScript Object Notation.

In this lesson, we are covering the following skill learning outcome:

- The candidate masters JavaScript Object Notation techniques and styles.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out work based on the needs of a junior JavaScript developer.

Introduction to Objects

In JavaScript, most things are Objects, starting from arrays and ending on the browser's APIs built on top of JavaScript. In this section, we'll learn the syntax of a basic Object.

We must declare the Object with curly brackets. Put the property/method names within brackets and their values after the colon. Properties can have various values, such as strings, numbers or other Objects. Methods have properties containing functions as the values.

Let's create an example.html file:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <script>
```

```

const myObject = {
  name: "value",
  numberName: 20,
  smallObject: {
    smallName1: 30,
    smallName2: 30,
  },
  myFunc: function (status) {
    this.myStatus = status;
  },
};
</script>
</body>
</html>

```


In the example above, we created the new Object with properties: name, numberName, smallObject and a method, myFunc(). Notice that the myFunc() method uses the “this” keyword. This refers to the current Object. In other words, myFunc() sets the value of the myStatus property of myObject.

READ

1. Page: [Introducing JavaScript objects](#) by MDN Web Docs.
2. Page: [JavaScript object basics](#) by MDN Web Docs.

WATCH

1. Video: [Objects: A practical introduction](#) (4m 53s) by Morten Rand-Hendriksen on LinkedIn Learning.
2. Video: [JavaScript objects: The code version](#) (2m 57s) by Morten Rand-Hendriksen on LinkedIn Learning.

	<p>ACTIVITY</p> <p>Create an Object: matchScore with two properties: homeScore and awayScore. Both properties should be numbers. Object should also have the method homeGoal() and awayGoal(), which increments the value of the homeScore or awayScore.</p> <p>Answer: Click here to reveal.</p>
---	---

Object containers

JavaScript Objects can be seen as containers for variables and methods. In our previous example, we declared it using the **const** keyword. Due to this, we can't change the reference to the Object itself. However, we can still modify the Object properties.

Let's copy the JavaScript part of our example and paste it in our browser's console:

```
>      const myObject = {
        name: "value",
        numberName: 20,
        smallObject: {
          smallName1: 30,
          smallName2: 30,
        },
        myFunc: function (status) {
          this.myStatus = status;
        },
      };

< undefined
> myObject = 123
✖ ▶ Uncaught TypeError: Assignment to constant variable.
   at <anonymous>:1:10
> myObject.name = "New name"
< 'New name'
> myObject.name
< 'New name'
> |
```

As mentioned earlier, we can't assign anything to our myObject variable as it is a **const**. However, we can easily modify its properties.

Since Objects are addressed by reference, we can describe them as mutable.

```
const myObject2 = myObject;
```

The above statement won't create a new Object. It will only link the created Object to the new variable. Let's look at a more complex example:

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
```

```

<script>
  const myObject = {
    name: "value",
    numberName: 20,
    smallObject: {
      smallName1: 30,
      smallName2: 30,
    },
    myFunc: function (status) {
      this.myStatus = status;
    },
  };
  const myObject2 = myObject;
  myObject2.name = "New name";
</script>
</body>
</html>

```

We see that changing the name of myObject2 also changes the name of myObject:

```

> const myObject = {
  name: "value",
  numberName: 20,
  smallObject: {
    smallName1: 30,
    smallName2: 30,
  },
  myFunc: function (status) {
    this.myStatus = status;
  },
};
const myObject2 = myObject;
myObject2.name = "New name";
< 'New name'
> myObject.name
< 'New name'

```

WATCH

Video: [Object containers](#) (2m 22s) by Morten Rand-Hendriksen on LinkedIn Learning.

Object properties

Object properties are defined using a colon-separated name-value pair. The property name must be a valid JavaScript identifier and start with a letter, underscore or dollar sign. Subsequent characters can also be digits. We can pass almost anything as a value. For example, primitive types, arrays or even other objects.

Commas separate properties.

Good programming practice tells us to use Camel case in our property names. Camel case starts with a lowercase letter, and every following word begins with a capital letter, for example:

- simpleText
- textContainingManyWords
- isAProperName

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <script>
      const myObject = {
        name: "value",
        numberName: 20,
        smallObject: {
          smallName1: 30,
          smallName2: 30,
```

```

        },
        myFunc: function (status) {
            this.myStatus = status;
        },
    };
</script>
</body>
</html>

```

In our example, we notice that commas separate all properties, and property names use Camel case, such as `numberName` or `smallObject`.

WATCH

Video: [Object properties \(55s\)](#) by Morten Rand-Hendriksen on LinkedIn Learning.

Accessing objects

We already know what objects properties are. In the previous examples, we could access them using dot notation: `object.propertyName`. We can also use dot notation to access nested objects: `outerObject.innerObject.propertyName`.

Dot notation is the fastest and most popular option, but there are other ways to access object properties.

We can use bracket notation to access object properties. It has the following syntax:

- `object["property"]` – for example: `myObject["name"]` from the previous example
- or `object[expression]` – for example:

```

function getPropertyName() {
    return "name";
}

```

```
var x = getPropertyName();  
myObject[x];
```

Of course, the function returning the property name could be much more complex. For example, we could use prompts to get the property name from the user.

We can also use bracket notation to access the nested objects. For example: `outerObject["innerObject"]["propertyName"]`.

We aren't obliged to stick to one notation. We can mix them. Instead of `outerObject.innerObject.propertyName` or `outerObject["innerObject"]["propertyName"]`, we can access this property name using:

`outerObject.innerObject["propertyName"]` or `outerObject["innerObject"].propertyName` accessors.

To add a new property to an existing object, all we need to do is give a value to the property name we want to create, for example:

```
const myObject = {  
  name: "value",  
  numberName: 20,  
  smallObject: {  
    smallName1: 30,  
    smallName2: 30,  
  }  
};  
myObject.numberName2 = 30;
```

The code above will add the `numberName2` property to the `myObject`. Ensure the property you want to create doesn't already exist in the Object.

We need to use the "delete" keyword to delete an existing property. After this keyword, we need to specify the name of the property we want to delete using either dot or bracket notation.

Let's look at an example:

```
const myObject = {  
  name: "value",  
  numberName: 20,  
  smallObject: {
```

```
        smallName1: 30,  
        smallName2: 30,  
    }  
};  
delete myObject.numberName;
```

The code above creates new objects and removes numberName properties.

WATCH

1. Video: [Accessing objects](#) (2m 32s) by Morten Rand-Hendriksen on LinkedIn Learning.
2. Video: [Accessing object properties](#) (5m 35s) by Morten Rand-Hendriksen on LinkedIn Learning.

ACTIVITY

Create an empty object.

Create a website containing two buttons:

- Add property – adds property named “new” to our Object, sets its value to 10 and displays the Object in the console.
- Remove property – Removes the “new” property from the Objects and displays the object in the console.

Answer: Click here to reveal.

Object methods

Object methods are object properties that are functions. We can access them using dot notation: `myObject.myMethod()`.

If we access `myObject.myMethod`, i.e. specify the object function name only and leave the parenthesis off the end of the name; we'll get the function definition instead. Let's look at an example:

```
const obj = {  
  name: "name",  
  func: function () {  
    return "text";  
  }  
};
```



```
> const obj = {  
      name: "name",  
      func: function () {  
        return "text";  
      }  
    };  
  
< undefined  
> obj.func()  
< 'text'  
> obj.func  
< f () {  
      return "text";  
    }  
> |
```

We can add methods after creating the Object, similar to adding any other property. We don't have to use our own functions. We can use built-in ones:

```
const obj = {};  
obj.name = name;
```



```
obj.bigName = function () {  
    return this.name.toUpperCase();  
}
```

In this case, we used the built-in `toUpperCase` function in a `bigName` method that we added to our `obj` object.

We used the “`this`” keyword in our method. The “`this`” keyword refers to an Object. The Object depends on how “`this`” is being invoked. Alone, it refers to the Global Object (we’ll learn about this later in the course). However, in our case – when invoked within our created Object – it refers to this Object. Because of this, we can use it to access other properties within object methods.

WATCH

Video: [Object methods](#) (6m 29s) by Morten Rand-Hendriksen on LinkedIn Learning.

ACTIVITY

Create an Object containing one number property called “`value`” and two methods: `increment()` and `decrement()`, which increments/decrements this value (adds or removes one from it). Display the value in the console logs.

Create a website containing two buttons: Increment and Decrement. Link Object methods to the button.

Answer: [Click here to reveal.](#)

What did I learn in this lesson?

This lesson provided the following insights:

- Objects and how to create them.
- Object properties and how to modify them.
- Object methods.

References

MDN Web Docs (2022) *JavaScript*. Available at: <https://developer.mozilla.org/en-US/docs/Web/JavaScript>

1.1. Lesson task - Introduction to Objects (part 1)

The task

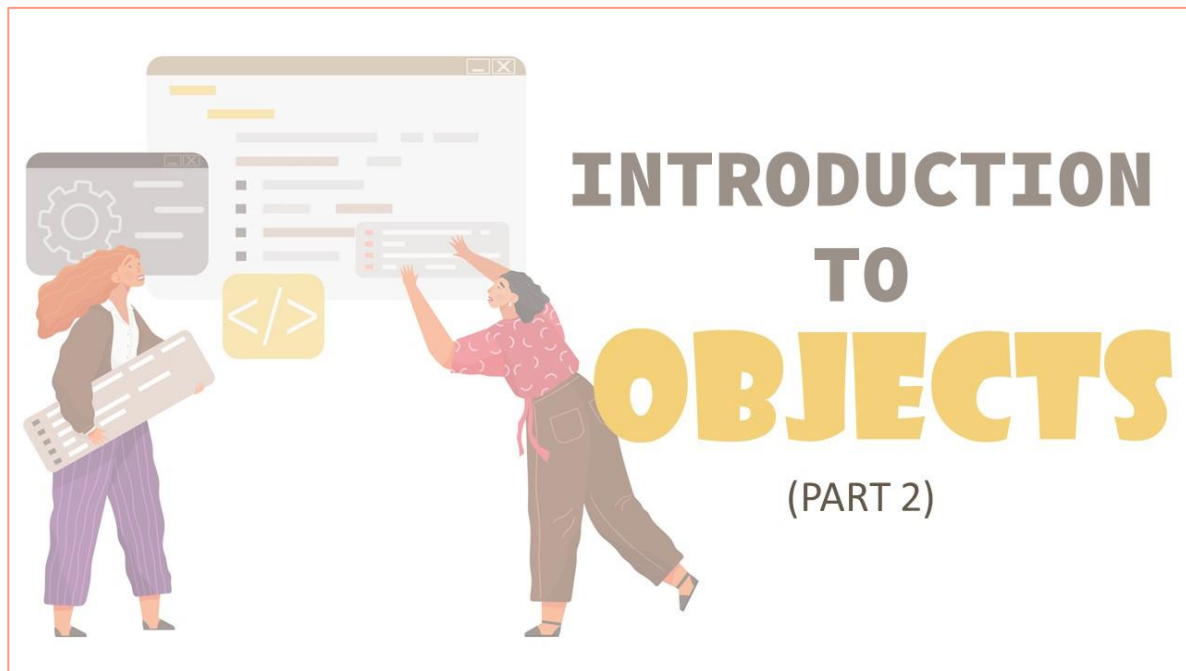
In this lesson, we learned about Object basics, such as Object properties and methods.

In this task:

- Create an Object containing two number properties and two methods showing the sum and multiplication of these properties in the console.
- Create a button which acts as a user for a property name. If the provided name is the same as the name of one of the number properties, ask them for a new value for it.
- Create two buttons and link the object method to them.

1.2. Lesson - Introduction to Objects (part 2)

Introduction



We already learned Object basics, such as Object properties and methods. In this lesson, we'll move to more advanced topics.

We also learned about classes. Now, we'll look at them as Objects and learn about Object constructors, prototypes and iterables.

Finally, we'll go through the most popular Global Objects. Due to Global Objects, we can access various built-in functionalities connected to, for example, math or date operations. We've often used them subconsciously and will now discuss them.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of concepts, processes and tools in Object Orientated programming and JavaScript Object Notation.

In this lesson, we are covering the following skill learning outcome:

- The candidate masters JavaScript Object Notation techniques and styles.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out work based on the needs of a junior JavaScript developer.

Classes as objects

We've already learned how to create and modify Objects. We can create them flexibly, as each can contain different properties and methods. While very convenient, it can lead to severe code readability issues during the design of an extensive application. We want our Objects to fit some rules to keep code readability at a proper level.

Again, let's look at myObject created in the previous lesson:

```
const myObject = {  
  name: "value",  
  numberName: 20,  
  smallObject: {  
    smallName1: 30,  
    smallName2: 30,  
  },  
  myFunc: function (status) {  
    this.myStatus = status;  
  }  
};
```

```
    },  
};
```

Let's create a class for Objects:

```
class MyClass {  
    constructor(  
        name,  
        numberName,  
        smallName1,  
        smallName2,  
        status  
    ) {  
        this.name = name;  
        this.numberName = numberName;  
        this.smallObject = {  
            smallName1: smallName1,  
            smallName2: smallName2  
        }  
        this.myStatus = status;  
    }  
    myFunc(statusValue) {  
        this.myStatus = statusValue;  
    }  
}
```

As mentioned in the previous lessons, classes also have their methods and properties. Both code examples have similar results. Let's create an Object of class MyClass and compare it to myObject after calling myFunc with "status" text as the parameter:

```

> myObject.myFunc("status")
< undefined

> myObject
< {name: 'value', numberName: 20, smallObject: {...}, myStatus: 'status', myFunc: f} ⓘ
  ▶ myFunc: f (status)
    myStatus: "status"
    name: "value"
    numberName: 20
  ▶ smallObject: {smallName1: 30, smallName2: 30}
  ▶ [[Prototype]]: Object
>

```

```

> var newObject = new MyClass("value", 20, 30, 30, "status")
< undefined

> newObject
< MyClass {name: 'value', numberName: 20, smallObject: {...}, myStatus: 'status'} ⓘ
  ▶ myStatus: "status"
  ▶ name: "value"
  ▶ numberName: 20
  ▶ smallObject: {smallName1: 30, smallName2: 30}
  ▶ [[Prototype]]: Object
>

```

The main differences are:

- object of MyClass has the class name displayed
- methods (myFunc()) aren't displayed as properties for the object of MyClass.

READ

Page: [Classes in JavaScript](#) by MDN Web Docs.

WATCH

Video: [Classes: Object blueprints](#) (6m 35s) by Morten Rand-Hendriksen on LinkedIn Learning.

ACTIVITY

Design a class for the following Object:

```
const obj = {  
  value1: 1,  
  value2: 2,  
  sum: function () {  
    console.log(this.value1 + this.value2);  
  },  
  multiplication: function () {  
    console.log(this.value1 * this.value2);  
  },  
};
```

Answer: [Click here to reveal.](#)

Object constructors

Classes are extremely useful and widely used in JavaScript development. However, there's a faster way to create templates for our Objects, which relies only on the basic function. This is called an Object Constructor function. It's similar to the class constructor, but we also declare methods.

Let's look at an example of the same Object represented by this function:

```
function MyFunction (  
  name,  
  numberName,
```

```

    smallName1,
    smallName2,
    status
) {
    this.name = name;
    this.numberName = numberName;
    this.smallObject = {
        smallName1: smallName1,
        smallName2: smallName2
    }
    this.myStatus = status;
    this.myFunc = function(statusValue) {
        this.myStatus = statusValue;
    }
}

```

```

> var obj = new MyFunction("value", 20, 30, 30, "status")
< undefined
> obj
< ▼ MyFunction {name: 'value', numberName: 20, smallObject: {...}, myStatus: 'status', myFunc: f} ⓘ
  ▶ myFunc: f (statusValue)
    myStatus: "status"
    name: "value"
    numberName: 20
  ▶ smallObject: {smallName1: 30, smallName2: 30}
  ▶ [[Prototype]]: Object
>

```

We notice that methods are now visible as properties. On the other hand, Object still has the “class” name at its start. The rest of the properties is the same.

WATCH

Video: [Object constructors](#) (2m 21s) by Morten Rand-Hendriksen on LinkedIn Learning.

ACTIVITY

Use the Object constructor to create a type for the following Object:

```
const obj = {  
  value1: 1,  
  value2: 2,  
  sum: function () {  
    console.log(this.value1 + this.value2);  
  },  
  multiplication: function () {  
    console.log(this.value1 * this.value2);  
  },  
};
```

Answer: [Click here to reveal.](#)

Object prototypes

All JavaScript objects inherit properties and methods from a prototype. For example, Date Object inherits from Date.prototype, and Array Object from Array.prototype. The Object.prototype is at the top of the inheritance chain (all Objects inherit from it).

We just learned how to use the Object constructor. However, we can't add new properties to an existing object constructor without editing the constructor itself. Fortunately, we might be able to create some workaround using prototypes.

The JavaScript prototype property allows us to add new properties to object constructors. For example, for the following object constructor:

```
function MyFunction (  
  name,  
  numberName,  
  status  
) {  
  this.name = name;
```

```
    this.numberName = numberName;
    this.myStatus = status;
}
```

The line

`MyFunction.newProperty = "new"` won't do anything:

```
> function MyFunction (
    name,
    numberName,
    status
) {
    this.name = name;
    this.numberName = numberName;
    this.myStatus = status;
}
MyFunction.newProperty = "new";

< 'new'

> var obj = new MyFunction("value", 20, "status")
< undefined

> obj.newProperty
< undefined
```

On the other hand, `MyFunction.prototype.newProperty` will create a new property, even though it won't be visible in the console:

```

> function MyFunction (
    name,
    numberName,
    status
  ) {
    this.name = name;
    this.numberName = numberName;
    this.myStatus = status;
  }
  MyFunction.prototype.newProperty = "new";
< 'new'

> var obj = new MyFunction("value", 20, "status")
< undefined

> obj
< ▼ MyFunction {name: 'value', numberName: 20, myStatus: 'status'} ⓘ
  myStatus: "status"
  name: "value"
  numberName: 20
  ► [[Prototype]]: Object

> obj.newProperty
< 'new'

```

In summary, prototypes are a very powerful tool, but we must use them wisely. Remember, we should modify only our own prototypes. Never modify the prototypes of standard JavaScript Objects.

READ

Page: [Object prototypes](#) by MDN Web Docs.

Object iterables

Iterable Objects can be iterated with the for-of-loop. Arrays and Strings are examples:

```
for (const x of [1,2,3,4,5]) {  
  // code block to be executed  
}
```

We can also create our own iterable objects. To do this, we need to implement our own iterator. The iterator protocol defines how to generate a sequence of values from the Object. The Object becomes an iterator when it has the `next()` method implemented, which returns an object with two properties:

- `value` – the next value
- `done` – Boolean, true if the iterator is completed, false if it produced a new value.

To create a JavaScript Iterable, we need to assign the `Symbol.iterator` property with a function returning the `next()` function. It sounds more complex than it is. Let's look at an example:

```
numbers = {};  
numbers[Symbol.iterator] = function() {  
  let n = 0;  
  done = false;  
  return {  
    next() {  
      n += 2;  
      if (n == 10) {done = true}  
      return {value:n, done:done};  
    }  
  };  
}
```

First, we assign the start of iteration – “n” and “done” values as the global variables. We use them within the `next()` function, from which we return them. Let's look at this code in action:

```
> numbers = {};  
< ▶ {}  
  
> numbers[Symbol.iterator] = function() {  
  let n = 0;  
  done = false;  
  return {  
    next() {  
      n += 2;  
      if (n == 10) {done = true}  
      return {value:n, done:done};  
    }  
  };  
}  
for(var number of numbers) {  
  console.log(number);  
}  
  
2  
4  
6  
8  
  
< undefined
```

READ

Page: [Iteration protocols](#) by MDN Web Docs.

Global Objects

We've been using Global Objects all along. We just weren't aware. The term Global Objects refers to objects in the global scope. They can be accessed using the "this" keyword. Global Objects work the same as all the other objects we have learned about so far, with one exception – we don't need to define them.

Global Objects provide us with functionalities such as mathematical operations (Math object), date information (Date object) or even data structures (Map and Set objects).

You can read more in the official Mozilla documentation linked in the READ section.

Let's look at an example of using the Date object. You can see detailed documentation of this object on the Mozilla documentation linked in the READ section.

Let's create a Person object with name, surname and date of birth. We can use the Date object to get the person's age based on their birth date.

```
function Person (
    name,
    surname,
    dateOfBirth
) {
    this.name = name;
    this.surname = surname;
    this.dateOfBirth = dateOfBirth;
    this.currentAge = function() {
        var today = new Date();
        var birthDate = new Date(dateOfBirth);
        var age = today.getFullYear() -
birthDate.getFullYear();
        var m = today.getMonth() - birthDate.getMonth();
        if (m < 0 || (m === 0 && today.getDate() <
birthDate.getDate())) {
            age--;
        }
        return age;
    }
}
```

First, we set the properties in the constructor. The most interesting part is the currentAge() method. We get the current date by creating the new empty Date() object – it sets to today by default. We then create the second Date object – based on the date of birth.

The Date object can convert numerous date formats. Next, we set the number of full years of the person we created. If they haven't had a birthday this year, we subtract one from their age. We then return the result.

Let's look at this method in action:

```
> var obj = new Person("John", "Doe", "December 5, 1977 15:00:00 PST");  
< undefined  
> obj.currentAge()  
< 44  
> |
```

READ

1. Page: [Standard built-in objects](#) by MDN Web Docs.
2. Page: [Date](#) by MDN Web Docs.

WATCH

Video: [Global objects](#) (7m 47s) by Morten Rand-Hendriksen on LinkedIn Learning.

What did I learn in this lesson?

This lesson provided the following insights:

- How to design templates for the Objects using classes or Object constructors.
- Object prototypes and when to use them.

- Creating your own iterables.
- Using Global Objects.

References

MDN Web Docs (2022) *Web technology for developers*. Available at:
<https://developer.mozilla.org/en-US/docs/Web>

1.2. Lesson task - Introduction to Objects (part 2)

The task

In this lesson, we learned about more complex object details. In this task, you need to design two object templates for the following Object:

```
const dog = {  
    name: "Rex",  
    age: 5,  
    bark: function () {  
        console.log("Bark bark!")  
    },  
};
```

Using the Class and Object constructor, create a new method birth() which returns the year of the dog's birth.

1.3. Lesson - JSON (part 1)

Introduction



JSON (JavaScript Object Notation) is a popular format for sharing data online. Its structure is based on JavaScript syntax. However, it isn't only for working with JavaScript but numerous other technologies.

JSON is used widely in data exchange, such as API requests and responses or configuration files. In the past, XML language was used in these areas, even though JSON is significantly easier for humans to read. It also needs fewer characters to code the data, which can save resources while working with massive data.

In this lesson, we'll learn JSON basics, such as JSON rules, structure, conversion and the most popular libraries supporting it. This subject is split into two lessons. The more complex details of JSON are addressed in the next lesson.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of concepts, processes and tools in Object Orientated programming and JavaScript Object Notation.

In this lesson, we are covering the following skill learning outcome:

- The candidate masters JavaScript Object Notation techniques and styles.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out work based on the needs of a junior JavaScript developer.

Introduction to JSON

JSON is a modern text format for storing and transporting data. It's self-describing and easy to understand for humans. Even though it's based highly on JavaScript Object structure, it's also entirely language-independent. Code for reading and generating JSON exists in many programming languages.

Let's assume we have the following Object:

```
{  
  name: 'John',  
  age: 30,  
  car: null  
}
```

The JSON equivalent will be very similar. The main difference is that property names are written within double quotations, as are string values. Below we can see what the JSON equivalent looks like:

```
{
  "name": "John",
  "age": 30,
  "car": null
}
```

There are more syntax rules, including:

- No leading zeros in number values.
- No trailing decimals in number values – for example, “2.” is wrong.
- No trailing commas – in JavaScript, we can always put a comma at the end of the last property, for example:

```
{
  name: 'John',
  age: 30,
  car: null,
}
```

is valid in JavaScript. In JSON, it is not.

READ

1. Page: [JSON](#) by MDN Web Docs.
2. Page: [Introducing JSON](#) by JSON.

WATCH

1. Video: [What is JSON? \(1m 49s\)](#) by Sasha Vodnik on LinkedIn Learning.
2. Video: [Structure JSON by following the rules \(4m 7s\)](#) by Sasha Vodnik on LinkedIn Learning.

Building JSON structure

To create a JSON, we need to create a .json file first. We'll save our data in this file. JSON is only plain text, but we can create and edit it in any text editor. More advanced editors, such as Visual Studio Code, offer JSON support, such as colouring the font.

Let's create an obj.json file and paste the JSON from the previous example.

```
{  
  "name": "John",  
  "age": 30,  
  "car": null  
}
```

We can also have collections using JSON Arrays. We must put our data within square brackets to create a JSON array. Then we don't need to specify keys:

```
[  
  "John",  
  30,  
  null  
]
```

As we can see, a JSON array can contain various data types.

In practice, JSON objects are usually much more complex. Objects are often nested. Let's create a simple nested JSON object:

```
{
  "array1": [
    {
      "value1": 10,
      "value2": "dog",
      "value3": true
    },
    "data"
  ],
  "value4": null,
  "array2": [
    "data",
    "anotherData",
    123
  ]
}
```

Our JSON object contains arrays and properties. The first array contains another object and property.

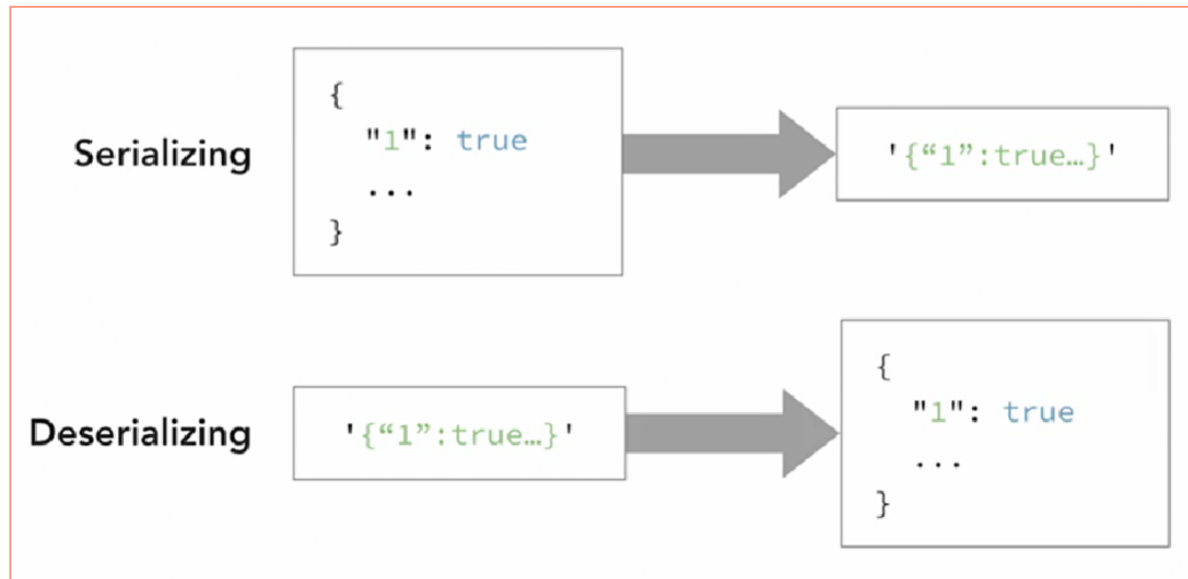
WATCH

1. Video: [Build a basic JSON structure \(3m 50s\)](#) by Sasha Vodnik on LinkedIn Learning.
2. Video: [Build a nested JSON structure \(2m 45s\)](#) by Sasha Vodnik on LinkedIn Learning.

JSON libraries and programming tools

JSON is only the text – we can't use this data directly in our programmes. We need to convert it to the format supported by our programming language. Fortunately, most modern programming languages provide multiple JSON tools.

The process of producing new JSON is called serialization. The process of getting data from JSON is deserialization:



In JavaScript, we can use the `JSON.parse()` and `JSON.stringify()` methods to manage it. The `stringify()` method is responsible for serialization, while the `parse()` method is responsible for deserialization.

Let's look at an example of how to use them. We can also learn more in the Mozilla documentation linked in the READ section:

```
const json = '{"result":true, "count":42}';
const obj = JSON.parse(json);
console.log(obj.count);
// expected output: 42
console.log(obj.result);
// expected output: true
console.log(JSON.stringify({ x: 5, y: 6 }));
// expected output: '{"x":5,"y":6}'
console.log(JSON.stringify([new Number(3), new
String('false'), new Boolean(false)]));
// expected output: '[3,"false",false]'
console.log(JSON.stringify({ x: [10, undefined, function(){},
Symbol('')] }));
// expected output: '{"x":[10,null,null,null]}'
console.log(JSON.stringify(new Date(2006, 0, 2, 15, 4, 5)));
// expected output: '"2006-01-02T15:04:05.000Z'"
```

There are also some other useful JSON tools:

- [JSON compare](#) - shows the difference between two JSONs.
- [JSON beautifier](#) - formats our JSON to improve the readability.
- [JSON validator](#) - checks if our JSON is valid.

READ

1. Page: [JSON.parse\(\)](#) by MDN Web Docs.
2. Page: [JSON.stringify\(\)](#) by MDN Web Docs.

WATCH

Video: [Process JSON with language tools and libraries](#) (1 m 18s) by Sasha Vodnik on LinkedIn Learning.

JSON conversion

This section will focus on the complex uses of serialization and deserialization methods. Let's get the JSON data from the API request:

api_example.js:

```
$.ajax ({
  url:
  'https://api.github.com/repos/your_github_username/myfirstrep
o',
  success: function(response) {
    console.log(response)
  }
})
```

jquery.html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>App</title>
    <script src="jquery-3.6.0.js"></script>
    <script src="api_example.js"></script>
  </head>
</html>
```

We must remember that the jQuery library automatically converts JSON to JavaScript objects, so the displayed response will be the JavaScript object. We can check the JSON in the browser's console. For example, after entering the Network tab, we should see the request "myfirstrepo". Click it and move to the "Response" tab:

Our JSON should look like this:

```
{
  "id": 517017220,
  "node_id": "R_kgDOHtEOhA",
  "name": "MyFirstRepo",
  "full_name": "[REDACTED]/MyFirstRepo",
  "private": false,
  "owner": {
    "login": "[REDACTED]",
    "id": 109850860,
    "node_id": "U_kgDOBoww7A",
    "avatar_url": "https://avatars.githubusercontent.com/u/109850860?v=4",
    "gravatar_id": "",
    "url": "https://api.github.com/users/[REDACTED]",
    "html_url": "https://github.com/[REDACTED]",
    "followers_url": "https://api.github.com/users/[REDACTED]/followers",
    "following_url": "https://api.github.com/users/[REDACTED]/following",
    "subscriptions_url": "https://api.github.com/users/[REDACTED]/subscriptions",
    "organizations_url": "https://api.github.com/users/[REDACTED]/orgs",
    "repos_url": "https://api.github.com/users/[REDACTED]/repos",
    "events_url": "https://api.github.com/users/[REDACTED]/events",
    "received_events_url": "https://api.github.com/users/[REDACTED]/received_events",
    "type": "User",
    "site_admin": false
  },
  "created_at": "2022-01-10T10:10:10Z",
  "updated_at": "2022-01-10T10:10:10Z",
  "pushed_at": "2022-01-10T10:10:10Z",
  "git_refs_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/git/refs",
  "git_tags_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/git/tags",
  "git_commits_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/git/commits",
  "branches_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/branches",
  "tags_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/tags",
  "files_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/files",
  "blobs_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/git/blobs",
  "pages_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/pages",
  "statuses_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/statuses/[REDACTED]",
  "language": null,
  "has_watcher": true,
  "watchers_count": 1,
  "subscription_url": "https://api.github.com/repos/[REDACTED]/MyFirstRepo/subscription",
  "permissions": {
    "admin": false,
    "push": true,
    "pull": true
  }
}
```

We can see its JavaScript equivalent logged in the console.

Let's serialize this object again, display in the console log, deserialize and display in logs:

```
$.ajax ({
  url:
'https://api.github.com/repos/user_name/myfirstrepo',
  processData: false,
  success: function(response) {
    console.log(response)
```



```

const serialized = JSON.stringify(response);
console.log(serialized);
const deserialized = JSON.parse(serialized);
console.log(deserialized);
}
})

```

After looking in the logs, we notice that JSONs are less readable than JavaScript objects:

```

{
  "id": 517017220, "node_id": "R_kgDOHtEOHA", "name": "MyFirstRepo", "full_name": "Bartektrr/MyFirstRepo", "private": false, "owner": {
    "login": "Bartektrr", "id": 109850860, "node_id": "U_kgDOB0w7A", "avatar_url": "https://avatars.githubusercontent.com/u/109850860?v=4", "gravatar_id": "", "url": "https://api.github.com/users/Bartektrr", "html_url": "https://github.com/Bartektrr", "followers_url": "https://api.github.com/users/Bartektrr/followers", "following_url": "https://api.github.com/users/Bartektrr/following", "gists_url": "https://api.github.com/users/Bartektrr/gists", "starred_url": "https://api.github.com/users/Bartektrr/starred", "owned_repos_url": "https://api.github.com/users/Bartektrr/repos", "subscriptions_url": "https://api.github.com/users/Bartektrr/subscriptions", "organizations_url": "https://api.github.com/users/Bartektrr/orgs", "repos_url": "https://api.github.com/users/Bartektrr/repos", "events_url": "https://api.github.com/users/Bartektrr/events", "received_events_url": "https://api.github.com/users/Bartektrr/received_events", "type": "User", "site_admin": false, "html_url": "https://github.com/Bartektrr/MyFirstRepo", "description": null, "fork": false, "url": "https://api.github.com/repos/Bartektrr/MyFirstRepo", "forks_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/forks", "keys_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/keys", "collaborators_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/collaborators", "teams_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/teams", "hooks_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/hooks", "issue_events_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/issues/events", "events_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/events", "assignees_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/assignees", "branches_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/branches", "tags_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/tags", "blobs_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/git/blobs", "git_tags_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/git/tags", "git_refs_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/git/refs", "trees_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/git/trees", "statuses_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/statuses", "languages_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/languages", "stargazers_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/stargazers", "contributors_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/contributors", "subscribers_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/subscribers", "subscription_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/subscription", "commits_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/commits", "git_commits_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/git/commits", "comments_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/comments", "issue_comment_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/issues/comments", "contents_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/contents", "compare_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/compare", "merges_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/merges", "archive_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/archive", "downloads_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/downloads", "issues_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/issues", "pulls_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/pulls", "milestones_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/milestones", "notifications_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/notifications", "releases_url": "https://api.github.com/repos/Bartektrr/MyFirstRepo/releases", "created_at": "2022-07-23T09:43:41Z", "updated_at": "2022-08-14T16:38:50Z", "pushed_at": "2022-07-24T02:25:24Z", "ssh_url": "git@github.com:Bartektrr/MyFirstRepo.git", "clone_url": "https://github.com/Bartektrr/MyFirstRepo.git", "svn_url": "https://github.com/Bartektrr/MyFirstRepo", "homepage": null, "size": 10, "stargazers_count": 0, "watchers_count": 0, "language": "HTML", "has_issues": true, "has_projects": true, "has_downloads": true, "has_wiki": true, "has_pages": false, "forks_count": 0, "mirror_url": null, "archived": false, "disabled": false, "open_issues_count": 0, "license": null, "allow_forking": true, "is_template": false, "web_commit_signoff_required": false, "topics": [], "visibility": "public", "forks": 0, "open_issues": 0, "watchers": 0, "default_branch": "main", "temp_clone_token": null, "network_count": 0, "subscribers_count": 1}
}

```

JavaScript objects take only one line in the console, and we can expand them if we want to. In contrast, JSON is fully displayed as plain text. Because of this, we might need to use the tools mentioned earlier, which improve JSON's readability.

WATCH

1. Video: [Convert JSON to an object or array \(4m 58s\)](#) by Sasha Vodnik on LinkedIn Learning.
2. Video: [Convert an object or array to JSON \(3m 37s\)](#) by Sasha Vodnik on LinkedIn Learning.

JSON and other data formats

JSON is the most popular data format, but XML (Extensible Markup Language) was mostly used before JSON gained popularity. In the future, we might have to work with various data formats within one project. Fortunately, there are tools to convert data from one format to another.

You might need to find the proper library for yourself, depending on which formats you want to convert. We'll use the `jsonxml` library, which helps with converting between JSON and XML formats. We can download the [jsonxml library here](#).

We can look at an example of how to use it in the `xmljson_test.html` file:

```
<html>
<head>
<title> XML/JSON - Tests </title>
<script type="text/javascript" src="xml2json.js"></script>
<script type="text/javascript" src="json2xml.js"></script>
<script type="text/javascript">

var xml=[
    '<e/>',

    '<e>text</e>',

    '<e name="value" />',

    '<e name="value">text</e>',

    '<e> <a>text</a> <b>text</b> </e>',

    '<e> <a>text</a> <a>text</a> </e>',

    '<e> text <a>text</a> </e>',

    '<a>hello</a>',

    '<a x="y">hello</a>',

    '<a id="a"><b id="b">hey!</b></a>'
```

```

'<a>x<c/>y</a>',

'<x u=""/>',

'<html>' +
' <head>' +
' <title>Xml/Json</title>' +
' <meta name="x" content="y" />' +
' </head>' +
' <body>' +
' </body>' +
'</html>',

'<ol class="xoxo">' +
' <li>Subject 1' +
' <ol>' +
' <li>subpoint a</li>' +
' <li>subpoint b</li>' +
' </ol>' +
' </li>' +
' <li><span>Subject 2</span>' +
' <ol compact="compact">' +
' <li>subpoint c</li>' +
' <li>subpoint d</li>' +
' </ol>' +
' </li>' +
'</ol>',

'<span class="vevent">' +
' <a class="url" href="http://www.web2con.com/">' +
' <span class="summary">Web 2.0 Conference</span>' +
' <abbr class="dtstart" title="2005-10-05">October
5</abbr>' +
' <abbr class="dtend" title="2005-10-08">7</abbr>' +
' <span class="location">Argent Hotel, San
Francisco, CA</span>' +
' </a>' +
'</span>',

```

```

'<e>\n'+
'  <![CDATA[ .. some data .. ]]>\n'+
'</e>',

'<e>\n'+
'  <a />\n' +
'  <![CDATA[ .. some data .. ]]>\n'+
'  <b />\n' +
'</e>',

'<e>\n'+
'  some text\n' +
'  <![CDATA[ .. some data .. ]]>\n'+
'  more text\n' +
'</e>',

'<e>\n'+
'  some text\n' +
'  <![CDATA[ .. some data .. ]]>\n'+
'  <a />\n' +
'</e>',

'<e>\n'+
'  <![CDATA[ .. some data .. ]]>\n'+
'  <![CDATA[ .. more data .. ]]>\n'+
'</e>'
];

function parseXml(xml) {
  var dom = null;
  if (window.DOMParser) {
    try {
      dom = (new DOMParser()).parseFromString(xml,
"text/xml");
    }
    catch (e) { dom = null; }
  }
  else if (window.ActiveXObject) {
    try {

```

```

        dom = new ActiveXObject('Microsoft.XMLDOM');
        dom.async = false;
        if (!dom.loadXML(xml)) // parse error ..
            window.alert(dom.parseError.reason +
dom.parseError.srcText);
    }
    catch (e) { dom = null; }
}
else
    alert("oops");
return dom;
}

window.onload = function() {
    var json;
    eval('a = {"e":null}');
    for (var i=0; i<xml.length; i++) {
        show(xml[i] + "\n\n" +
            (json = xml2json(parseXml(xml[i]), " ")) + "\n\n"
+
            json2xml(eval('json='+json))));
    }
}


function show(s) { document.getElementById("out").innerHTML
+= (s+"\n").replace(/&/g,
"&").replace(/</g, "<").replace(/>/g, ">").replace(/\n/g,
"<br/>") + "<hr/>"; }


</script>
</head>

<body>
<pre id="out"></pre>
</body>
</html>

```

We use the `xml2json()` method to convert XML format to JSON format and `json2xml()` to get XML format from the JSON.

	<p>READ</p> <p>Article: Converting Between XML and JSON by Stefan Goessner.</p>
---	--

	<p>WATCH</p> <p>Video: Convert between JSON and other data formats (4m 43s) by Sasha Vodnik on LinkedIn Learning.</p>
---	--

What did I learn in this lesson?

This lesson provided the following insights:

- JSON and why it is important.
- How to create JSON objects and operate on the ones already created.
- What libraries and tools can improve our experience of working with JSON.

1.3. Lesson task - JSON (part 1)

The task

In this lesson, we learned about standard JSON basics. You must create a JSON based on the JavaScript object in this task.

```
{  
  name: 'John',  
  age: 30,  
  car: null  
}
```

Display the JSON in the console.

Convert the JSON to the XML using the `json2xml()` method in the `jsonxml` library and display it in the console.

1.4. Lesson - JSON (part 2)

Introduction



In the previous lesson, we learned about JSON basics. We already know how to create JSON objects, why they are important and how to convert them to other data types. This lesson will focus on more advanced details, such as various types and conventions of creating JSONs.

Working with JSON objects isn't always easy. They can be enormous and organised differently than we expect. JSON Schema is a standard for creating schemas which work as JSON object's documentation. We'll learn how to create and generates our own schemas.

The last topic we'll learn about in this lesson is JSON for Linked Data (JSON-LD). This is based on the JSON format but adds more rules, so data is organised better. Since this format is prevalent, there's a significant chance that external JSON objects will also be in this format.

Learning outcomes

In this lesson, we are covering the following knowledge learning outcome:

- The candidate has knowledge of concepts, processes and tools in Object Orientated programming and JavaScript Object Notation.

In this lesson, we are covering the following skill learning outcome:

- The candidate masters JavaScript Object Notation techniques and styles.

In this lesson, we are covering the following general competence learning outcome:

- The candidate can carry out work based on the needs of a junior JavaScript developer.

Applying techniques for working with JSON

We already know we can use the `JSON.stringify()` method to serialize our JavaScript objects to JSON. However, precisioning the parameters will make the JSON more readable for humans:

```
JSON.stringify(value, replacer, spacer)
```


- value – the object we want to serialize
- replacer – the function that modifies how the conversion happens
- spacer – customisation of the indents for the levels hierarchy.

Let's look at an example:

json_example.js:

```
const jsonData = '{"name": "John", "age": 30, "car": null}';
console.log(jsonData);
const object = JSON.parse(jsonData);
console.log(JSON.stringify(object, null, 5));
```

```
{"name": "John", "age": 30, "car": null}
{
    "name": "John",
    "age": 30,
    "car": null
}
```

First, we create the JSON, then convert it to a JavaScript object and back to JSON with the spacer parameter. On the screen, we notice how the JSON displayed improved heavily from a human perspective.

Usually, the empty values are “falsy”, not “truthy”. For example, after comparing an empty string to the false with the “==” operator, we would get information that they’re equal.

An empty object { } is “truthy”. We can see this in an example below:

```
> const myObject = {};
< undefined
> if (myObject) {
    console.log("Empty object is truthy.");
}
Empty object is truthy.
< undefined
```

The if-condition was fulfilled. Because of this, we might run into bugs when we compare an empty object to Booleans or null and get not expected result.

While working with modern browsers, we can use the `Object.keys()` method to get the number of object properties. However, it isn't enough, as some non-empty built-in objects might not be empty. We need to check if our data is an Object.

Let's look at example of testing whether the object is empty or not:

```
<!DOCTYPE html>
<html>
<head>
  <title>Testing for an empty object</title>
</head>
<body>
  <script>
    const emptyObject = {};
    const nonEmptyObject = { name: "value" };
    const stringContent = "string";
    const numberContent = 123;
    const arrayContent = [ "data1", "data2", "data3" ];
    const dateContent = new Date();

    const testData = (data) => {
      if (Object.keys(data).length === 0 &&
data.constructor === Object) {
        console.log('Empty Object:');
        return data;
      } else {
        console.log('Not an empty object:');
        return data;
      }
    }
  </script>
</body>
</html>
```

We can run the function `testData()` with the object name as the parameter to check if this object is empty. We should use it whenever we're accessing data from

external sources to check if our request was successful and if we got the data we expected.

```
> testData(nonEmptyObject)
Not an empty object:
```

READ

1. Page: [JSON.stringify\(\)](#) by MDN Web Docs.
2. Page: [Object.keys\(\)](#) by MDN Web Docs.

WATCH

1. Video: [Return readable JSON](#) (3m 25s) by Sasha Vodnik on LinkedIn Learning.
2. Video: [Test for an empty object](#) (6m 46s) by Sasha Vodnik on LinkedIn Learning.

Introduction to JSON Schema

While working with data, we won't only use the data we created ourselves. Sometimes, we might have to use external data. Since it's sometimes challenging to understand other people's ideas, understanding the way the external data is designed also isn't easy.

JSON Schema enables us to create documentation of our existing data formats so that other people can read and understand them. It's also responsible for validating the data. Here, we can define our object details, for example, which properties are required or their expected data type.

Let's create a simple JSON Schema. We can see details of how to do it on the official JSON Schema website (linked in the READ section). However, it will be easier to show this in an example:

schema.js

```
var animalsSchema = {
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "http://zooWebsite.com/schemas/animals.json",
  "title": "Animals",
  "description": "Schema for animals in the local zoo",
  "type": "array",
  "items": {
    "type": "object",
    "properties": {
      "name": {
        "type": "string"
      },
      "weight": {
        "type": "string"
      },
      "age": {
        "type": "number"
      }
    }
  }
};
```

This schema describes the Animals object. Even though schema uses only JSON to describe the data, it's easier to store it in files of our programming language. In our case, JavaScript files. Every schema has some characteristic properties:

- `$schema` – defines the standard in which the schema is written – linked to the official JSON Schema website.
- `$id` – unique URI. Usually, we want to provide the name of the domain we control here. The URL doesn't have to work.
- Title and description – annotation keywords, descriptive only. They are here to help human users understand the data.
- Type – validation, defines the first constraint on our JSON data.

We'll learn how to create more detailed schemas in the next section.

	<p>READ</p> <ol style="list-style-type: none"> 1. Page: Getting Started Step-By-Step by JSON Schema. 2. Page: Specification by JSON Schema.
--	--

	<p>WATCH</p> <ol style="list-style-type: none"> 1. Video: What is JSON Schema? (2m 57s) by Sasha Vodnik on LinkedIn Learning. 2. Video: Create a basic schema with JSON Schema (6m 47s) by Sasha Vodnik on LinkedIn Learning.
--	--

Working with JSON Schema

After creating the schema, we must ensure it's valid and working as expected. There are numerous JSON Schema validators. We'll use the [online one](#). We only need to pass the JSON object on the Select Schema tab. If there are any errors, they'll be displayed at the bottom of the page. For example, if we copied the entire code, we copied the schema and the variable declaration, and we'll get a proper error about it.

After pasting the correct Schema, we can create examples of JSON objects and test if they fit the Schema:

Select schema: Custom

1 {
2 "\$schema": "http://json-schema.org/draft-07/schema#",
3 "\$id": "http://zoowebsite.com/schemas/animals.json",
4 "title": "Animals",
5 "description": "Schema for animals in the local zoo",
6 "type": "array",
7 "items": {
8 "type": "object",
9 "properties": {
10 "name": {
11 "type": "string"
12 },
13 "weight": {
14 "type": "string"
15 },
16 "age": {
17 "type": "number"
18 }
19 }
20 }
21 }
22

Input JSON: ✖ Found 1 error(s)

1 [{"name": "Pig", "age": "abc"}] |

✖ Found 1 error(s)

Message: **Invalid type. Expected Number but got String.**
Schema path: <http://zoowebsite.com/schemas/animals.json#/items/properties/age/type>

We need to pass the array of animal objects in the specified Schema. We can see the error - the age property of our animal isn't correct. It has a string type, not the number, as specified in the Schema. After changing the age property value to a number, there are no more errors.

As we can see in the example above, not all properties have to be specified now. We can also fix this in the "required" property at the end of the schema. We need to put all required property names within the array. Let's make all properties required:

Select schema: Custom

1 {
2 "\$schema": "http://json-schema.org/draft-07/schema#",
3 "\$id": "http://zoowebsite.com/schemas/animals.json",
4 "title": "Animals",
5 "description": "Schema for animals in the local zoo",
6 "type": "array",
7 "items": {
8 "type": "object",
9 "properties": {
10 "name": {
11 "type": "string"
12 },
13 "weight": {
14 "type": "string"
15 },
16 "age": {
17 "type": "number"
18 }
19 },
20 "required": [
21 "name",
22 "weight",
23 "age"
24]
25 }
26 }
27

Input JSON: ✖ Found 1 error(s)

1 [{"name": "Pig", "age": 22}]

JSON Schema significantly improves our JSON object's readability, but creating them can be time-consuming. Fortunately, we can use the JSON Schema

generators to create them quicker. We need to pass JSON examples, and the generator will create the JSON Schema for us.

Please check out one of the most common [JSON Schema generators](#).

Using this generator for the following JSON object:

```
[{
  "name": "Pig",
  "age": 22
}, {
  "name": "Cow",
  "age": 23
}]
```

We got the Schema:

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "http://example.com/example.json",
  "type": "array",
  "default": [],
  "title": "Root Schema",
  "items": {
    "type": "object",
    "title": "A Schema",
    "required": [
      "name",
      "age"
    ],
    "properties": {
      "name": {
        "type": "string",
        "title": "The name Schema",
        "examples": [
          "Pig",
          "Cow"
        ]
      },
      "age": {
```

```

        "type": "integer",
        "title": "The age Schema",
        "examples": [
            22,
            23
        ]
    },
    "examples": [{
        "name": "Pig",
        "age": 22
    },
    {
        "name": "Cow",
        "age": 23
    }
    ],
    "examples": [
        [{
            "name": "Pig",
            "age": 22
        },
        {
            "name": "Cow",
            "age": 23
        }
    ]
    ]
}

```

The generated Schema isn't perfect yet. We need to customise titles or IDs, for example. However, we now only need to edit a few properties of the already existing Schema instead of creating an entire Schema from scratch.

WATCH

1. Video: [Validate JSON data against a schema \(3m 42s\)](#) by Sasha Vodnik on LinkedIn Learning.
2. Video: [Specify required properties with JSON Schema \(4m 59s\)](#) by Sasha Vodnik on LinkedIn Learning.
3. Video: [Work with schema generators \(7m 29s\)](#) by Sasha Vodnik on LinkedIn Learning.

JSON-LD

JSON-LD stands for JSON for Linked Data. This is the implementation format for structuring data. It uses the schema.org vocabulary, a joint effort of such companies as Google, Yahoo!, Bing and Yandex, to create unified data vocabulary for the web. By adding JSON-LD to our websites, we can improve how it's displayed in the searchers, which increases the chance of getting new users.

Let's see how to create a simple JSON-LD snippet for our JSON company object:

```
{
  "company": "my Company",
  "website": "https://mycompany.com",
  "overview": "This is my small company. It has been
created recently, so it is still in the development phase."
}
```

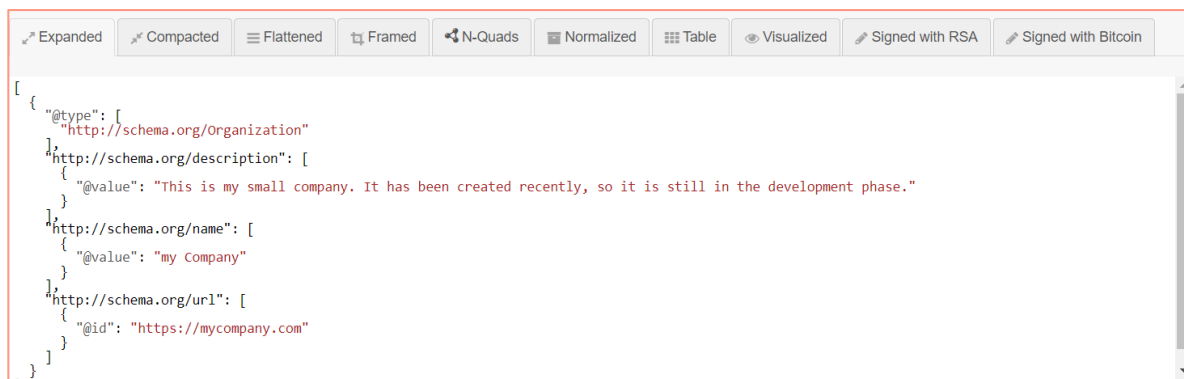
First, let's open the [JSON-LD Playground page](#) and click the "Person" button. We can check out the other buttons as well. All snippets contain the @context property, linking to the [Schema.org page](#). This page contains documentation on what properties we can use and their types.

We need to define the @type property as well. After looking at the documentation, we notice that the organisation type is the closest one for us.

Let's look at the documentation, search for property names for the rest of our data, and then change them. Our final object should look like this:

```
{
  "@context": "http://schema.org/",
  "@type": "Organization",
  "name": "my Company",
  "url": "https://mycompany.com",
  "description": "This is my small company. It has been
created recently, so it is still in the development phase."
}
```

After the property name changes, this object finally fits the standard of JSON-LD, f
Having a valid JSON-LD snippet, we can paste it on [JSON-LD Playground](#) and see how it's presented:



JSON-LD is simple to implement within our websites. All we need to do is place the JSON-LD script in the head section of our website.

If we had a company's website, we could include the file above in the head section of the home page:

```
<script type="application/json+ld">
  {
    "@context": "http://schema.org/",
    "@type": "Organization",
    "name": "my Company",
    "url": "https://mycompany.com",
    "description": "This is my small company. It has
been created recently, so it is still in the development
phase."
```

```
}  
</script>
```

Thanks to this, our website will be better accessed by search engines.

READ

1. Website: <https://json-ld.org/>
2. Website: <https://schema.org/>
3. Website: <https://www.w3.org/TR/json-ld/>
4. Article: [A Guide to JSON-LD for Beginners](#) by Alexis Sanders.

WATCH

1. Video: [What is JSON-LD?](#) (2m 46s) by Sasha Vodnik on LinkedIn Learning.
2. Video: [Implement predefined key names](#) (5m 33s) by Sasha Vodnik on LinkedIn Learning.
3. Video: [Add a JSON-LD snippet to a web page](#) (2m 46s) by Sasha Vodnik on LinkedIn Learning.

What did I learn in this lesson?

This lesson provided the following insights:

- The techniques we can apply while working with JSON.
- How to describe our JSON objects with JSON Schema.
- JSON-LD.

References

MDN Web Docs (2022) *Web technology for developers*. Available at:
<https://developer.mozilla.org/en-US/docs/Web>

1.4. Lesson task - JSON (part 2)

The task

This lesson taught us complex information about JSON, such as JSON Schemas and JSON-LD.

In this task, you need to create a JSON Schema for the following object:

```
{
  "checked": false,
  "dimensions": {
    "width": 5,
    "height": 10
  },
  "id": 1,
  "name": "A green door",
  "price": 12.5,
  "tags": [
    "home",
    "green"
  ]
}
```

Each property is required.

Try to create the schema yourself and then check the schema generated by
<https://www.jsonschema.net>. Compare them.

1.5. Lesson - Self-study

Introduction



This is a self-study lesson which consolidates knowledge of the first module. It contains videos reminding parts of working with JavaScript Objects and JSON.

Materials

WATCH

1. Video: [Introduction to JavaScript objects](#) (5m 6s) by Engin Arslan on LinkedIn Learning.
2. Video: [Object methods and 'this' keyword](#) (6m 20s) by Engin Arslan on LinkedIn Learning.
3. Video: [Constructor functions](#) (8m 43s) by Engin Arslan on LinkedIn Learning.
4. Video: [JSON basics](#) (2m 45s) by Peter Gruenbaum on LinkedIn Learning.
5. Video: [JSON examples](#) (6m 14s) by Peter Gruenbaum on LinkedIn Learning.
6. Video: [Working with JavaScript Object Notation \(JSON\)](#) (6m 42s) by Jonathan Fernandes on LinkedIn Learning.

1.5. Lesson task - Self-study

The task

Complete a series of small tasks:

1. Having the following object:

```
const person = {  
  firstName: "John",  
  lastName: "Doe"  
};
```

Display the last name of the person in an alert.

Answer: Click here to reveal.

2. Create an object “people”, which consists of two properties, person objects, which have properties firstName and lastName (both Strings).

Answer: Click here to reveal.

3. Create a “person” object with name = John, surname = Doe and age = 50.

Access the object and display all the properties in an alert.

Answer: Click here to reveal.

4. Design a template for the following object:

```
const book = {  
  title: "Harry Potter",  
  releaseYear: 1997,  
  author: "J.K. Rowling",  
  cast: function() {  
    return alert("Avada Kedavra")  
  }  
};
```

using class.

Answer: Click here to reveal.

5. Design a template for the following object:

```
const book = {  
  title: "Harry Potter",  
  releaseYear: 1997,  
  author: "J.K. Rowling",  
  cast: function() {  
    return alert("Avada Kedavra")  
  }  
};
```

using an object constructor.

Answer: [Click here to reveal.](#)

6. Create a JSON Schema for the following object:

```
{
  "title": "Harry Potter",
  "releaseYear": 1997,
  "author": "J.K. Rowling",
  "spells": [
    "Avada Kedavra",
    "Expecto Patronum"
  ]
}
```

Answer: [Click here to reveal.](#)

7. Convert the following object to a JSON. Display it in the console with proper indents. Convert it back to the JavaScript object and display it in the console.

```
{
  title: "Harry Potter",
  releaseYear: 1997,
  author: "J.K. Rowling"
};
```

Answer: [Click here to reveal.](#)

8. Check the number of properties in the following objects:

```
const o1 = new Date();
const o2 = new String();
const o3 = new Number();
const o4 = {};
```

Check which of these are empty objects.

Answer: [Click here to reveal.](#)



**Noroff
Education**