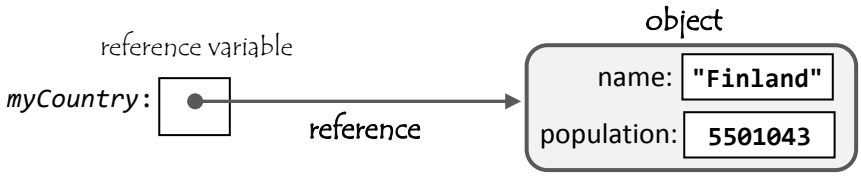
OBJECTS IN JAVASCRIPT / Short introduction only / Part 2

Using an object literal to create an object

This far, we have been creating objects by using object literals as below.

```
var myCountry = {name: "Finland", population: 5501043};
```



CREATING OBJECTS DYNAMICALLY

In object-oriented programming, you can define a "template" for creating objects. In JavaScript, we can *use a function as an **object constructor***[1].

First, we define an *object constructor function* as below. Normally, we start the function name with a capital letter. **NB!** We have to use the ***this*** keyword for referring to the properties.

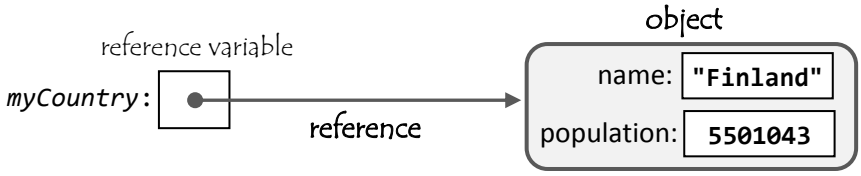
```
function Country(countryName, countryPopulation) {
    this.name = countryName;
    this.population = countryPopulation;
}
```

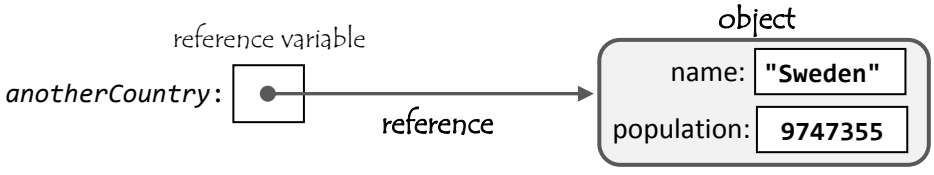
Any number of objects can be created with the ***new*** operator. This operator must be followed by the name of an object constructor function and the parameter list.

```
var myCountry = new Country("Finland", 5501043);
var anotherCountry = new Country("Sweden", 9747355);

console.log(myCountry.name + ", " + myCountry.population);
console.log(anotherCountry.name + ", " + anotherCountry.population);
```

In the above, the following objects are created:





[1] In JavaScript, there are ***several different ways to create objects***. In this handout, only one way is discussed. What is shown in this handout is not so far from the way how objects can be created in Java and C# as the other alternatives.

In the example below, we get the name and population of the country from the user.

```
var countryName = document.getElementById("txtCountryName").value;
var populationText = document.getElementById("txtPopulation").value;
var population = Number(populationText);
var myCountry = new Country(countryName, population);
```

Methods

Objects can encapsulate both ***data*** (several different data items in a single object) and ***behavior***. The behavior of an object is defined by defining one or more methods of the object. In JavaScript, a ***method*** is a JavaScript function that is invoked through an object.

In the example below, we define a method named ***toString***. This method makes it easy to get values of all properties of an object.

```
function Country(countryName, countryPopulation) {
    this.name = countryName;
    this.population = countryPopulation;

    this.toString = function() {
        return "name = " + this.name + ", population = " + this.population;
    }
}

var myCountry = new Country("Finland", 5501043);
console.log(myCountry.toString());
```

The code above prints the following to the console window:

```
name = Finland, population = 5501043
```

In the example below, we create three objects and save the object references in an array.

```
var total = 0;
var countryArray = [];
countryArray.push(new Country("Finland", 5501043));
countryArray.push(new Country("Sweden", 9747355));
countryArray.push(new Country("Norway", 5165802));

for (var i = 0; i < countryArray.length; i++) {
    total = total + countryArray[i].population;
    console.log(countryArray[i].toString());
}
console.log("The total of population is " + total);
```

The code above prints the following to the console window:

```
name = Finland, population = 5501043
name = Sweden, population = 9747355
name = Norway, population = 5165802
The total of population is 20414200
```

In the example below, we create an *object constructor function* for creating **Box** objects. There are three ***properties*** (*width, height, depth*) and three ***methods*** (*getSurfaceArea, getVolume, toString*).

```
function Box(boxWidth, boxHeight, boxDepth) {
    this.width = boxWidth;
    this.height = boxHeight;
    this.depth = boxDepth;

    this.getSurfaceArea = function() {
        var top = this.width * this.depth;
        var side = this.height * this.depth;
        var end = this.width * this.height;

        return (2 * top + 2 * side + 2 * end);
    }

    this.getVolume = function() {
        return (this.width * this.height * this.depth);
    }

    this.toString = function() {
        return "width = " + this.width +
            ", height = " + this.height +
            ", depth = " + this.depth +
            ", volume = " + this.getVolume() +
            ", surface area = " + this.getSurfaceArea();
    }
}
```

Then we use the object constructor function to create a Box ***object***.

```
var shoeBox = new Box(10, 5, 20);
```

We can print values of all properties of the Box object as follows:

```
console.log(shoeBox.toString());
```

The code above prints the following to the console window:

```
width = 10, height = 5, depth = 20, volume = 1000, surface area = 700
```

Here we print separately width, volume and surface area:

```
console.log("The width is " + shoeBox.width);

console.log("The volume is " + shoeBox.getVolume());

console.log("The surface area is " + shoeBox.getSurfaceArea());
```

The code above prints the following to the console window:

```
The width is 10
The volume is 1000
The surface area is 700
```

OPTIONAL ADVANCED TOPICS (NB! You can skip the below)

Nested objects

```
function City(cityName, cityPopulation) {
    this.name = cityName;
    this.population = cityPopulation;
}

function Country(countryName, countryPopulation, capitalCity, capitalPopulation) {
    this.name = countryName;
    this.population = countryPopulation;
    this.capital = new City(capitalCity, capitalPopulation);
}

var myCountry = new Country("Finland", 5501043, "Helsinki", 634509);

console.log("The population of " + myCountry.name +
                        " is " + myCountry.population);
console.log("The population of " + myCountry.capital.name +
                        " is " + myCountry.capital.population);
```

The code above prints the following to the console window:

```
The population of Finland is 5501043
The population of Helsinki is 634509
```

Improving performance

If you are creating a lot of objects of the same type, then the technique that we have been using to define methods has a shortcoming[2]. For example, when we create Country objects, *a copy of every method is physically added to each object*. JavaScript has a solution to this problem: it allows you to add methods to a *prototype object*.

In the example below, a copy of the ***toString*** method is ***not*** physically added to each Country object when they are created. Instead, the method is added to the ***Country prototype object*** only once and each Country object can use the *single physical copy* of the *toString* method of the prototype object.

```
function Country(countryName, countryPopulation) {
    this.name = countryName;
    this.population = countryPopulation;

    Country.prototype.toString = function() {
        return "name = " + this.name + ", population = " + this.population;
    }
}

var myCountry = new Country("Finland", 5501043);
console.log(myCountry.toString());
```

[2] For example, in Java and C#, we do not have this shortcoming.