Functional programming

Vicente Machaca Arceda

Javascript introduction

Javascript introduction

Local JavaScript Variables

```
// code here can not use carName

function myFunction() {
    var carName = "Volvo";

    // code here can use carName

}
```

Javascript introduction

Global JavaScript Variables

```
var carName = " Volvo";

// code here can use carName

function myFunction() {

    // code here can use carName

}
```

Javascript introduction

Automatically
Global

```
myFunction();

// code here can use carName

function myFunction() {
    carName = "Volvo";
}
```

Javascript introduction: var and let

```
function allyIlliterate() {
    //tuce is *not* visible out here

    for( let tuce = 0; tuce < 5; tuce++ ) {
        //tuce is only visible in here (and in the for() parentheses)
        //and there is a separate tuce variable for each iteration of the loop
    }

    //tuce is *not* visible out here
}

function byE40() {
    //nish *is* visible out here

    for( var nish = 0; nish < 5; nish++ ) {
        //nish is visible to the whole function
    }

    //nish *is* visible out here
}
```

Javascript introduction: Objects

```
//una manera de crear objetos
var person = {firstName:"John", lastName:"Doe", age:50, eyeColor:"blue"};

//otra manera de crear objetos
var person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

Javascript introduction: Objects

```
///////////////constructores ////////////////////////////
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
    this.name = function() {return this.firstName + " " + this.lastName;};
}

var p1 = new Person("Vicente", "Machaca", 29, "cafe");
var p2 = new Person("Enrique", "Arceda", 29, "cafe");
```

Functional programming

Functional programming

Just because your program contains functions does not necessarily mean that you are doing functional programming.

Functional programming

Functional programming distinguishes between pure and impure functions. It encourages you to write pure functions.

Pure functions

Pure functions

A pure function must satisfy both of the following properties:

- **Referential transparency:** The function always gives the same return value for the same arguments. This means that the function cannot depend on any mutable state.
- **Side-effect free:** The function cannot cause any side effects. Side effects may include I/O (e.g., writing to the console or a log file), **modifying a mutable object**, reassigning a variable, etc.

Pure functions: Example

```
function multiply(a, b) {
  return a * b;
}
```

Pure functions: Example

Impure functions

```
let heightRequirement = 46;

// Impure because it relies on a mutable (reassignable) variable.
function canRide(height) {
  return height >= heightRequirement;
}

// Impure because it causes a side-effect by logging to the console.
function multiply(a, b) {
  console.log('Arguments: ', a, b);
  return a * b;
}
```

Pure functions

Benefits of pure functions:

- They're easier to reason about and debug because they don't depend on mutable state.
- The return value can be cached or "memorized" to avoid recomputing it in the future.
- They're easier to test because there are no dependencies (such as logging, Ajax, database, etc.) that need to be mocked.

Pure functions: Immutability

We looked at the **canRide**function. We decided that it is an impure function, because the **heightRequirement** could be reassigned. Here is a contrived example of how it can be reassigned with unpredictable results:

```
let heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}

// Every half second, set heightRequirement to a random number between 0 and 200.
setInterval(() => heightRequirement = Math.floor(Math.random() * 201), 500);

const mySonsHeight = 47;

// Every half second, check if my son can ride.
// Sometimes it will be true and sometimes it will be false.
setInterval(() => console.log(canRide(mySonsHeight)), 500);
```

Pure functions: Immutability

Let me reemphasize that captured variables do not necessarily make a function impure. We can rewrite the **canRide** function so that it is pure by simply changing how we declare the **heightRequirement** variable.

```
const heightRequirement = 46;

function canRide(height) {
  return height >= heightRequirement;
}
```

```
const constants = {
  heightRequirement: 46,
  // ... other constants go here
};
```

Pure functions: Immutability

The object can be mutated. As the following code illustrates, to gain true immutability, you need to prevent the variable from being reassigned, and you also need immutable data structures. The JavaScript language provides us with the **Object.freeze** method to prevent an object from being mutated.

```
'use strict';

// CASE 1: The object is mutable and the variable can be reassigned.
let o1 = { foo: 'bar' };

// Mutate the object
o1.foo = 'something different';

// Reassign the variable
o1 = { message: "I'm a completely new object" };
```

Pure functions: Immutability

```
// CASE 2: The object is still mutable but the variable cannot be reassigned.
const o2 = { foo: 'baz' };

// Can still mutate the object
o2.foo = 'Something different, yet again';

// Cannot reassign the variable
// o2 = { message: 'I will cause an error if you uncomment me' }; // Error!
```

Pure functions: Immutability

```
// CASE 3: The object is immutable but the variable can be reassigned.
let o3 = Object.freeze({ foo: "Can't mutate me" });

// Cannot mutate the object
// o3.foo = 'Come on, uncomment me. I dare ya!'; // Error!

// Can still reassign the variable
o3 = { message: "I'm some other object, and I'm even mutable -- so take that!" };
```

Pure functions: Immutability

```
// CASE 4: The object is immutable and the variable cannot be reassigned. This is what we want!!!!!!!!
const o4 = Object.freeze({ foo: 'never going to change me' });

// Cannot mutate the object
// o4.foo = 'talk to the hand' // Error!

// Cannot reassign the variable
// o4 = { message: "ain't gonna happen, sorry" }; // Error
```

Pure functions: Immutability

Immutability pertains to all data structures

```
const a = Object.freeze([4, 5, 6]);

// Instead of: a.push(7, 8, 9);
const b = a.concat(7, 8, 9);

// Instead of: a.pop();
const c = a.slice(0, -1);
```

Pure functions: Immutability

Immutability pertains to all data structures

```
const map = new Map([
  [1, 'one'],
  [2, 'two'],
  [3, 'three']
]);

// Instead of: map.set(4, 'four');
const map2 = new Map([...map, [4, 'four']]);

// Instead of: map.delete(1);
const map3 = new Map([...map].filter(([key]) => key !== 1));

// Instead of: map.clear();
const map4 = new Map();
```

What is the problem?

Pure functions: Immutable.js

```
// Use in place of `[]`.
const list1 = Immutable.List(['A', 'B', 'C']);
const list2 = list1.push('D', 'E');

console.log([...list1]); // ['A', 'B', 'C']
console.log([...list2]); // ['A', 'B', 'C', 'D', 'E']


// Use in place of `new Map()`
const map1 = Immutable.Map([
  ['one', 1],
  ['two', 2],
  ['three', 3]
]);
const map2 = map1.set('four', 4);

console.log([...map1]); // [['one', 1], ['two', 2], ['three', 3]]
console.log([...map2]); // [['one', 1], ['two', 2], ['three', 3], ['four', 4]]
```

Function composition

Function composition

$$(f \circ g)(x) = f(g(x))$$

Function composition

```
// h(x) = x + 1
// number -> number
function h(x) {
  return x + 1;
}

// g(x) = x^2
// number -> number
function g(x) {
  return x * x;
}

// f(x) = convert x to string
// number -> string
function f(x) {
  return x.toString();
}
```

```
// y = (f ∘ g ∘ h)(1)
const y = f(g(h(1)));
console.log(y); // '4'
```

Function composition: With Ramda

```
function h(x) {
  return x + 1;
}

// g(x) = x^2
// number -> number
function g(x) {
  return x * x;
}

// f(x) = convert x to string
// number -> string
function f(x) {
  return x.toString();
}
```

```
// R = Ramda
// composite = (f ∘ g ∘ h)
const composite = R.compose(f, g, h);

// Execute single function to get the result.
const y = composite(1);
console.log(y); // '4'
```

Function composition: With Ramda

Okay, so we can do function composition in JavaScript. What's the big deal?

Function composition: With Ramda

Okay, so we can do function composition in JavaScript. What's the big deal?

Well, if you're really onboard with functional programming, then ideally your entire program will be nothing but function composition. There will be no loops (**for**, **for...of**, **for...in**, **while**, **do**) in your code.

Recursion

Recursion

Implement factorial function.

Recursion

```
function iterativeFactorial(n) {
  let product = 1;
  for (let i = 1; i <= n; i++) {
    product *= i;
  }
  return product;
}
```

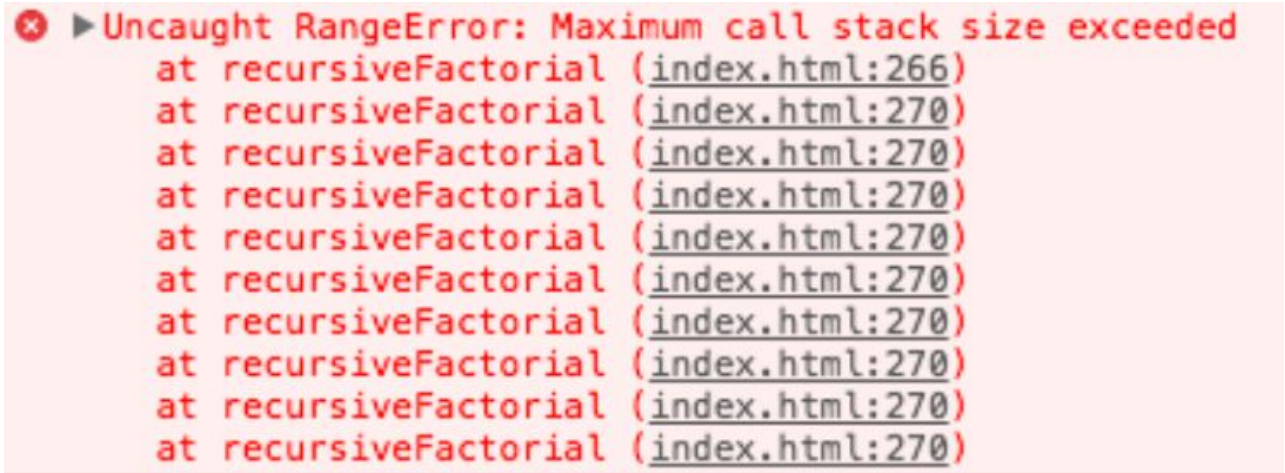
```
function recursiveFactorial(n) {
  // Base case -- stop the recursion
  if (n === 0) {
    return 1; // 0! is defined to be 1.
  }
  return n * recursiveFactorial(n - 1);
}
```

Recursion

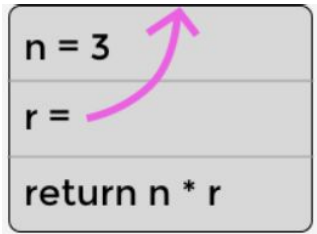
compute **recursiveFactorial(20000)**

Recursion

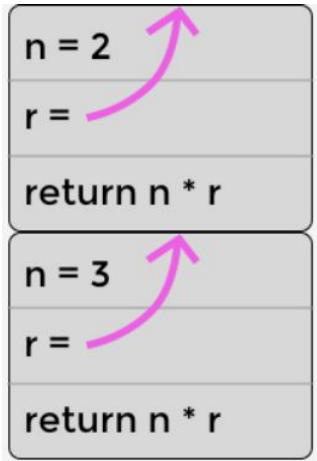
compute **recursiveFactorial(20000)**



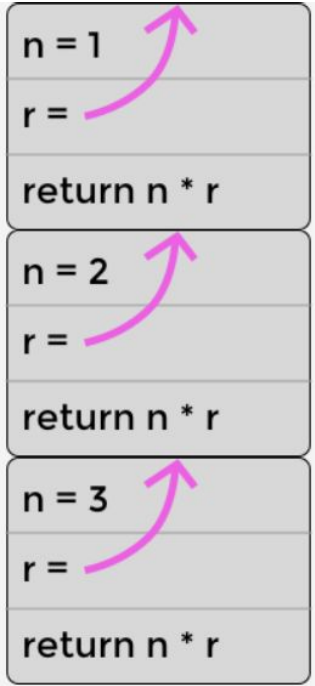
Recursion



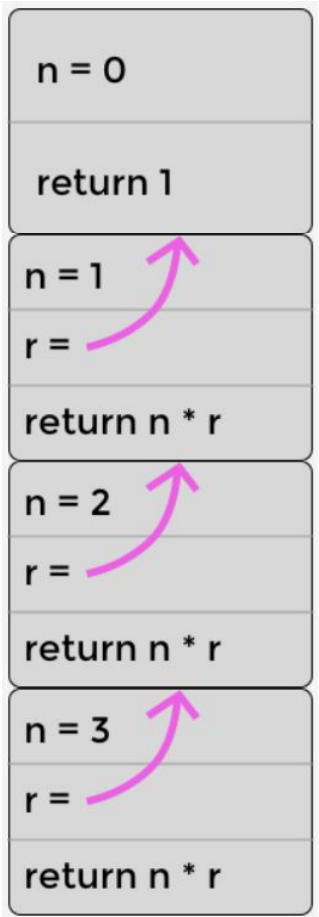
Recursion



Recursion



Recursion



Recursion

As part of the **ES2015** (aka **ES6**) specification, an optimization was added to address this issue. It's called the *proper tail calls optimization* (**PTC**). It allows the browser to elide, or omit, stack frames if the last thing that the recursive function does is call itself and return the result.

Recursion

We can rewrite the function in such a way so that the last step is the recursive call.

```
// Optimized for tail call optimization.
function factorial(n, product = 1) {
  if (n === 0) {
    return product;
  }
  return factorial(n - 1, product * n)
}
```

Recursion

Safari is the only browser that has implemented PTC.

Solution: Syntactic Tail Calls (STC)

Recursion

STC

```
function factorial(n, acc = 1) {
  if (n === 1) {
    return acc;
  }

  return continue factorial(n - 1, acc * n)
}
```

Higher-order functions

Higher-order functions

Functions that can be passed around just like any other value. We can pass a function to another function. We can also return a function from a function.

Higher-order functions

Functions can assign to variables

```
f = (m) => console.log(m)
f('Test')
```

Higher-order functions

Functions can assign to objects

```
obj = {
      f(m) {
          console.log(m)
      }
    }
    obj.f('Test')
```

Higher-order functions

Functions can assign to arrays

```
a = [
      m => console.log(m),
      m => console.log(m + "_jejeje")
    ]
    a[0]('Test')
    a[1]('Test')
```

Higher-order functions

Can be return by other functions

```
const createF = () => {
        return (m) => console.log(m)
}

const f = createF()
f('Test')
```

DECLARATIVE VS IMPERATIVE

An imperative approach is when you tell the machine (in general terms), the steps it needs to take to get a job done.

A declarative approach is when you tell the machine what you need to do, and you let it figure out the details.

DECLARATIVE VS IMPERATIVE

For example a declarative programming approach is to avoid using loops and instead use functional programming constructs like ***map***, ***reduce*** and ***filter***, because your programs are more abstract and less focused on telling the machine each step of processing.

Higher-order functions II

Higher-order functions: Example MAP

```
var numbers= [2, 4, 6, 8], counter;
var numbers2 = [];
```

//write a program that assign the *numbers2* vector with
//elements of vector *numbers* pow to 2

Higher-order functions: Example MAP

```
var numbers= [2, 4, 6, 8], counter;
var numbers2 = [];

//write a program that assign the numbers2 vector with
//elements of vector numbers pow to 2

for(counter = 0; counter < numbers.length; counter++){
        numbers2.push(numbers[counter]*2);
}
console.log(numbers2);
```

Higher-order functions: Example MAP

```
var numbers= [2, 4, 6, 8], counter;
var numbers2 = [];

//write a program that assign the numbers2 vector with
//elements of vector numbers pow to 2

var numbers3 = numbers.map(x => x*2);
console.log(numbers3);
```

Higher-order functions: Example MAP

```
var numbers= [2, 4, 6, 8], counter;
var numbers2 = [];
```

//write a program that assign the *numbers2* vector with
//elements *even* or *odd*

Higher-order functions: Example MAP

```
var numbers= [2, 4, 6, 8], counter;
var numbers2 = [];

//write a program that assign the numbers2 vector with
//elements even or odd


 var numbers4 = numbers.map(x =>{if(x%2 == 0) return
 "par"; else return "impar";});
       console.log(numbers4);
```

Higher-order functions: Array prototype

The prototype constructor allows you to add new properties and methods to the Array() object.

```
Array.prototype.myUcase = function() {
        for (i = 0; i < this.length; i++) {
            this[i] = this[i].toUpperCase();
        }
    };

var fruits = ["Banana", "Orange", "Apple", "Mango"];
fruits.myUcase();
console.log(fruits);
```

Higher-order functions: Array prototype

The prototype constructor allows you to add new properties and methods to the Array() object.

//Using *array prototype*, write your version of *map* method
//for Arrays

Higher-order functions: Array prototype

The prototype constructor allows you to add new properties and methods to the Array() object.

//Using *array prototype*, write your version of *map* method //for Arrays

```
my map function on arrays
Array.prototype.myMap = function(f1) {
        newArray = [];
        for (i = 0; i < this.length; i++) {
                newArray.push(f1(this[i]));
        }
        return newArray;
};
var numbers5 = numbers.myMap(x => x*2);
```

Higher-order functions: Filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction',
'present'];

const result = words.filter(word => word.length > 6);
console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

Higher-order functions: Filter

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

```
function isBigEnough(value) {
	return value >= 10;
}

var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
// filtered is [12, 130, 44]
```

Higher-order functions: Filter

Write a program that filter invalids json ids

```
var arr = [
  { id: 15 },
  { id: -1 },
  { id: 0 },
  { id: 3 },
  { id: 12.2 },
  { },
  { id: null },
  { id: NaN },
  { id: 'undefined' }
];
```

Higher-order functions: Reduce

The reduce() method executes a reducer function (that you provide) on each member of the array resulting in a single output value.

```
const array1 = [1, 2, 3, 4];
const reducer = (accumulator, currentValue) => accumulator + currentValue;

console.log(array1.reduce(reducer));
// expected output: 10

console.log(array1.reduce(reducer, 5));
// expected output: 15
```

Higher-order functions: Reduce

The reduce() method executes a reducer function (that you provide) on each member of the array resulting in a single output value.

```
var total = [ 0, 1, 2, 3 ].reduce(
  ( accumulator, currentValue ) => accumulator + currentValue,  0
);
```

Higher-order functions: Reduce

Write a program that show the mean price of type car = ‘suv’

```
const vehicles = [
        { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
        { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
        { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
        { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
        { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
        { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
        { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
        { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
        { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
        { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
      ];
```

Higher-order functions

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];

const averageSUVPrice = vehicles
  .filter(v => v.type === 'suv')
  .map(v => v.price)
  .reduce((sum, price, i, array) => sum + price / array.length, 0);

console.log(averageSUVPrice); // 33399
```

Higher-order functions

Functional

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];
```

```
// Using `pipe` executes the functions from top-to-bottom.
const averageSUVPrice1 = R.pipe(
  R.filter(v => v.type === 'suv'),
  R.map(v => v.price),
  R.mean
)(vehicles);

console.log(averageSUVPrice1); // 33399

// Using `compose` executes the functions from bottom-to-top.
const averageSUVPrice2 = R.compose(
  R.mean,
  R.map(v => v.price),
  R.filter(v => v.type === 'suv')
)(vehicles);

console.log(averageSUVPrice2); // 33399
```

Higher-order functions

```
const vehicles = [
  { make: 'Honda', model: 'CR-V', type: 'suv', price: 24045 },
  { make: 'Honda', model: 'Accord', type: 'sedan', price: 22455 },
  { make: 'Mazda', model: 'Mazda 6', type: 'sedan', price: 24195 },
  { make: 'Mazda', model: 'CX-9', type: 'suv', price: 31520 },
  { make: 'Toyota', model: '4Runner', type: 'suv', price: 34210 },
  { make: 'Toyota', model: 'Sequoia', type: 'suv', price: 45560 },
  { make: 'Toyota', model: 'Tacoma', type: 'truck', price: 24320 },
  { make: 'Ford', model: 'F-150', type: 'truck', price: 27110 },
  { make: 'Ford', model: 'Fusion', type: 'sedan', price: 22120 },
  { make: 'Ford', model: 'Explorer', type: 'suv', price: 31660 }
];
```

It clearly separates the data (i.e., **vehicles**) from the logic (i.e., the functions **filter**, **map**, and **reduce**).

Functional

```
// Using `pipe` executes the functions from top-to-bottom.
const averageSUVPrice1 = R.pipe(
  R.filter(v => v.type === 'suv'),
  R.map(v => v.price),
  R.mean
)(vehicles);

console.log(averageSUVPrice1); // 33399

// Using `compose` executes the functions from bottom-to-top.
const averageSUVPrice2 = R.compose(
  R.mean,
  R.map(v => v.price),
  R.filter(v => v.type === 'suv')
)(vehicles);

console.log(averageSUVPrice2); // 33399
```

Currying

Currying

Currying is the process of breaking down a function into a series of functions that each take a **single argument**.

Currying

```
function dot(vector1, vector2) {
  return vector1.reduce((sum, element, index) => sum += element * vector2[index], 0);
}

const v1 = [1, 3, -5];
const v2 = [4, -2, -1];

// Use Ramda to do the currying for us!
const curriedDot = R.curry(dot);

const sumElements = curriedDot([1, 1, 1]);

console.log(sumElements(v1)); // -1
console.log(sumElements(v2)); // 1

// This works! You can still call the curried function with two arguments.
console.log(curriedDot(v1, v2)); // 3
```

Exercises

Exercise 1

It is ok?

```
function doAComputation(n1, n2) {
    if (n1 != 0)
        n1 = n1 + 1;
    if (n2 >  n3) {
        n2 = n2 - 1
        n3 = n3 + 5
    }
    return n1 + n2 + n3;
}

x = 3;
y = 4;
var result = doAComputation(x, y, z);
console.log(result);
```

Exercise 2

Implements function **trim**

Tips:

```
var x = “hola”;
var tipo = typeof x;  //tipo = “string”


x.lenght;  // lenght of string
x.charAt(0) // first character
```

Exercise 3

```
function extend(obj1, obj2) {
    // your code here
}

var bird1 = {
    name: "robin",
    songbird: true
};

var bird2 = {
    name: "ostrich",
    songbird: false,
    flies: false
};

var bird = {
    phylum: "chordata",
    class: "aves",
    feathered: true,
    flies: true
};

console.log("------ Extend with birds ------ ");
extend(bird1, bird);
console.log(bird1);

extend(bird2, bird);
console.log(bird2);
```

Write a function, **extend**, that takes two objects, and *extends* the first object with properties from the second object

Exercise 3

```
function extend(obj1, obj2) {
    // your code here
}

var bird1 = {
    name: "robin",
    songbird: true
};

var bird2 = {
    name: "ostrich",
    songbird: false,
    flies: false
};

var bird = {
    phylum: "chordata",
    class: "aves",
    feathered: true,
    flies: true
};

console.log("------ Extend with birds ------ ");
extend(bird1, bird);
console.log(bird1);

extend(bird2, bird);
console.log(bird2);
```

Write a function, **extend**, that takes two objects, and *extends* the first object with properties from the second object

```
function extend(obj1, obj2) {

    for (var prop in obj2) {
        if (obj1[prop] == undefined) {
            obj1[prop] = obj2[prop];
        }
    }
}
```

Exercise 4

Write a function, **objectContainsOnlyStrings**, that takes one object, and determines if that object contains only properties that have string values.

```
function objectContainsOnlyStrings(o) {
    // your code here
}
console.log("----- Contains Strings -----");

var bird = {
    phylum: "chordata",
    class: "aves",
    feathered: true,
    flies: true
};
console.log("Does bird contain only strings? " + objectContainsOnlyStrings(bird));

var oStrings = {
    s1: "string1",
    s2: "string2"
}
console.log("Does oStrings contain only strings? " + objectContainsOnlyStrings(oStrings));
```