CSEM: Programming JavaScript

Dr. Markus Klems, Prof. Stefan Tai

Wirtschaftsinformatik –
Information Systems Engineering

Technische Universität Berlin

PROGRAMMING JAVASCRIPT I

Learning objectives

- Understand variable scoping in JavaScript ES5 and ES2015
- Learn functional programming with JavaScript
  - Anonymous function expressions, ES2015 arrow function notation
  - Nested functions
  - Closures
  - Currying
- Learn basic Array and Array extras API

What is JavaScript?

- JavaScript is an untyped, interpreted programming language that is suited for functional and object-oriented programming.
- JavaScript is often referred to as “the programming language of the web” because a vast majority of dynamic web applications use JavaScript on the client-side.
  - In recent years, JavaScript has also gained popularity as a server-side programming language: Node.js framework (also known simply as “node”)
  - You can even build native mobile and desktop applications in JavaScript.
  - In a sense, JavaScript fulfills the “write once, run anywhere” promise of Java.

JavaScript “Flavors”

JavaScript (formally: ECMAScript) comes in different “flavors”

- Modern web browsers support ECMAScript 5.1 (ES5)
- Newer version that brings major improvements: ES2015 (aka ES6)
  - Browsers do not support ES2015
  - Compile/transpile ES2015 => ES5 with Babel
- TypeScript extends ES2015 with a type system and useful object-oriented programming concepts
  - Enables static verification and other developer-friendly features

JavaScript Expressions and Statements

Expressions

- An expression is a phrase of JavaScript code which resolves to a value.
- For example: assignment, function definition, arithmetic expression, etc.

Statements

- A statement is a phrase of JavaScript code which is executed.
- For example: declaration, control structures (loops, conditionals, jump)
- Expressions that are used to produce side-effects are known as expression statements, for example, an assignment expression statement.
- A JavaScript program is a bulk of statements that execute.

VARIABLES & MORE

Variables

JavaScript is untyped (no static types)

JavaScript
```
var guest = "Picard"; // guest is a string
guest = 1;            // guest is now a number
```

JavaScript distinguishes 2 categories of types:

- Primitive types (most important: `number`, `string`, `boolean`)
  - Primitive values are immutable
  - Primitives are compared *by value*
- Reference types (`array`, `function`, custom objects)
  - Reference type values are mutable
  - Reference types are compared *by reference*

Global scope

- JavaScript variables are global if they are declared outside of a function.
- Global variables can be accessed and manipulated from anywhere in your program.
- To be more precise, a global variable is a property of the *global object*.
- When a JavaScript engine starts (*e.g.*, web page reload), a new global object is created (in the web browser, the `window` object).

JavaScript

```javascript
var guest = "Picard";
var f = function() {
   guest = "Worf";
};
f();
console.log(guest);          // Worf
console.log(window.guest); // Worf
```

Local scope

- If a variable is declared with **var** inside of a function, the variable is local within the scope of that function.
- If you declare a variable without **var**, it is global, no matter where it has been declared.

JavaScript

```javascript
var f = function() {
   var hotel = "Vulcan Inn";
   nights = 3;
};
f();
console.log(nights); // 3
console.log(hotel); // ReferenceError: hotel is not defined
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Hoisting

- The scoping of variables in JavaScript has implications on the load order of your code.
- JavaScript uses “hoisting” to automatically move the declaration of variables to the top of a scope when a JavaScript program runs.

JavaScript

```javascript
var scope = 'G';

function f() {
 console.log(scope);
// undefined
 var scope = 'L';
 console.log(scope); // L
}


f();
```

JavaScript (hoisted)

```javascript
var scope = 'G';

function f() {
 var scope;
 console.log(scope);
// undefined
 scope = 'L';
 console.log(scope); // L
}

f();
```

Block scope (ES2015)

Using ES2015, you can bind a variable to the scope of a block by declaring the variable with `let` instead of `var` .

ES2015	JavaScript

```
var hotel = "Star Fleet Motel";

{
    let guest = "Geordi";
    let hotel = "Vulcan Inn";
}

console.log(guest);
// guest is not defined
```
	```
	var hotel = "Star Fleet Motel";

	{
	    var _guest = "Geordi";
	    var _hotel = "Vulcan Inn";
	}

	console.log(guest);
	// guest is not defined
	```

Constants

With `const`, you can create a read-only (immutable) constant. You can assign a value to a constant only once.

ES2015

```
const PLANETS = 9;
PLANETS = 8;
// TypeError: Assignment to constant variable.
```

Strict mode

You can tell the JavaScript engine to use more “foolproof” semantics than normal, the so-called “strict mode”. In strict mode, the JavaScript syntax is more restricted to hinder you from writing code that might cause common issues.

You can invoke strict mode either at the beginning of your script, i.e., before any other statement:

JavaScript

```javascript
"use strict";
// JavaScript code ...
```

Alternatively, you can invoke strict mode as the first statement inside of a function, if you only want the function to run in strict mode

JavaScript

```javascript
function() {
    "use strict";
    // JavaScript code ...
};
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

FUNCTIONS

Functions

- In JavaScript, a function is actually a special kind of object. Although JavaScript is not a functional language, this fact enables us to use functional programming techniques in JavaScript.
- A function can be defined with zero or more named parameters.
- The function body contains zero or more statements and all variables that are declared in a function body are function-scoped.
- You can use the `return` statement to return a value. If a function has no `return` statement, then the function returns the "absent value" `undefined`.

parameters

local variable

```
function trip(start, destination) {
    var toReturn = "Go on a trip from " + start +
      " to " + destination;
    return toReturn;
}
```

return statement

```
trip("Earth", "Mars");
// "Go on a trip from Earth to Mars"
```

Functions

- There are 2 ways to define a function: either via a function definition expression or via a function declaration statement.
- A function declaration statement takes the following form:

JavaScript

```javascript
function f() {
 console.log('function declaration statement');
}
```

- A function statement must have a name because the function name is used as a variable.
- A function definition expression takes the following form (the function name is optional):

JavaScript

```javascript
var f = function () {
 console.log('function definition expression');
};
```

Function statement hoisting

The fact that, in the case of a function statement, the named function is actually a variable, means that you need to be aware of function load order.

JavaScript	JavaScript (hoisted)

`function f() {`
` return 'A';`
`}`
 	
`var a = f();`
`console.log(a); // B`
 	
`function f() {`
` return 'B';`
`}`
 	
`var b = f();`
`console.log(b); // B`
	~~`function f() {`~~
	~~` return 'A';`~~
	~~`}`~~
	`function f() {`
	` return 'B';`
	`}`
	`var a;`
	`var b;`
	
	`a = f();`
	`console.log(a); // B`
	`b = f();`
	`console.log(b); // B`

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Anonymous Functions

You can create an anonymous function and assign it to a variable:

JavaScript

```
// Anonymous function that is assigned to variable f
var f = function () {
 console.log('foo');
};

f();
```

Anonymous Functions

- You can wrap an unnamed/anonymous function expression `function ()` inside a grouping operator, *i.e.*, a pair of parentheses `(...)`.
- Thereby, you can define and instantly invoke an anonymous function:

JavaScript

```
// Anonymous function that is instantly invoked
(function () {
 console.log('bar');
}());
```

Anonymous Functions

You need the grouping operator `(...)` to indicate to the JavaScript engine that you have written a function expression and not a function statement. Otherwise, you will get a syntax error, like in this example:

JavaScript

```
function () {
 console.log('this does not work');
}();
```

Inner (Nested) Functions

You can nest a function within a function. The inner function (aka “nested function”) can access the function-scope variables of the outer function. Consider the following example:

JavaScript

```javascript
function outer() {
 var scope1 = 'outer';
 return function inner() {
   var scope2 = 'inner';
   return scope1 + ' and ' + scope2 + ' scope';
 };
};

var result = outer()();
console.log(result); // inner and outer scope
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Inner (Nested) Functions

- JavaScript is a lexically scoped language
  - Variables that are defined within a function are local to that function (aka function-scoped).
  - Variables that are defined outside of a function, are global to the entire JavaScript program.
  - Functions within functions “inherit” the variable scope of their outer function, thereby forming a scope chain.

Inner (Nested) Functions

JavaScript

```
// Lexical scoping and scope chain
function outer() {
 var a = 'foo';
 return function inner() {
   return b + ' ' + a;
 };
};

var a = 'bar';
var b = 'kung';

console.log(outer()()); // ?
```

Inner (Nested) Functions

JavaScript

```javascript
// Lexical scoping and scope chain
function outer() {
 var a = 'foo';
 return function inner() {
   return b + ' ' + a;
 };
};

var a = 'bar';
var b = 'kung';

console.log(outer()()); // kung foo
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Closures

- A closure is a function that is bundled with its enclosing “environment”, *i.e.*, the scope chain of variables.
- In JavaScript, this is possible by encapsulating variables in the function scope of an outer variable (“environment”) and accessing them through an inner function.

JavaScript

```
// Closure
var max3 = (function () {
 var max = 3;
 return function () {
   if (max > 0) {
     return --max;
   } else {
     return 0;
   }
 };
}());
```

Closures

JavaScript

```
// Closure
var max3 = (function () {
 var max = 3;
 return function () {
   if (max > 0) {
     return --max;
   } else {
     return 0;
   }
 };
}());

console.log(max3() + ' are left.'); // 2 are left.
console.log(max3() + ' are left.'); // 1 are left.
console.log(max3() + ' are left.'); // 0 are left.
console.log(max3() + ' are left.'); // 0 are left.
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Closures

Another example of a closure is a “maker” function that can be used to construct multiple similar function instances. For example, the following `hotelMaker` closure is used to create hotel names for a particular hotel chain.

JavaScript

```javascript
function hotelMaker(hotelChain) {
 return function (location) {
     return hotelChain + ' in ' + location;
   };
}

var adlon = hotelMaker('Kempinski Adlon');
var hyatt = hotelMaker('Grand Hyatt');
var adlonBerlin = adlon('Berlin');
var hyattNYC = hyatt('New York City');
console.log(adlonBerlin);
console.log(hyattNYC);
```

Currying

This approach is generally known as “**currying**”, a concept that is similar to partial function application. Currying in JavaScript works like this: you split up a function that has many parameters into nested functions with fewer parameters.

JavaScript

```javascript
// original function
function hotelMaker(chain, planet, city) {
 return chain + ' ' + city + ', ' + planet;
}
var firstHotel = hotelMaker('Elon Inn', 'Earth', 'Berlin');
console.log(firstHotel);

// curried function
function curryHotelMaker(chain) {
 return function (planet) {
   return function (city) {
     return chain + ' ' + city + ', ' + planet;
   };
 };
}
var elonInn = curryHotelMaker('Elon Inn');
var secondHotel = elonInn('Mars')('Cydonia');
console.log(secondHotel);
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Arrow Functions

ES2015 introduces a shorter syntax for writing anonymous function expressions: the arrow function (“fat arrow”).

ES2015	JavaScript
`var welcome = (name) => 'Welcome ' + name;`	`var welcome = function (name) {` `return 'Welcome ' + name;` `};`
`console.log(welcome('Picard'));`	`console.log(welcome('Picard'));`

(Moreover, the arrow function lexically binds the `this` value which is useful when programming nested object methods.)

ARRAYS

Arrays

- A JavaScript array is a special kind of object.
- An array consists of zero or more ordered items, starting with index 0.
- You can access and manipulate items via the array index, *e.g.*, `hotels[2]`.

JavaScript

```
var hotels = ['Vulcan Inn', 'Star Fleet Motel', 'Borg
Bungalows', 'Klingon BnB'];
console.log("array length: " + hotels.length); // 4
console.log("get 3rd item: " + hotels[2]); // Borg Bungalows
hotels[2] = 'Romulus Resorts';
console.log("get 3rd item: " + hotels[2]); // Romulus Resorts
console.log("get index of 'Vulcan Inn': " +
hotels.indexOf('Vulcan Inn')); // 0
console.log("joined array: " + hotels.join(' and '));
// Vulcan Inn and Star Fleet Motel and Romulus Resorts and
Klingon BnB
```

Arrays: push() and pop()

- You can add an item to the end of an array with the `push(item)` method.
- You can remove the last item of an array with the `pop()` method.

ES2015

```
let hotels = ['Vulcan Inn', 'Star Fleet Motel', 'Romulus
Resorts', 'Klingon BnB'];

// push an item to the end of the array
hotels.push('Deep Space Nine');
console.log("array length: " + hotels.length); // 5
console.log("last item: " + hotels[4]); // Deep Space Nine

// remove an item from the end of the array
var ds9 = hotels.pop();
console.log("array length: " + hotels.length); // 5
console.log("last item: " + hotels[3]); // Klingon BnB
console.log("removed item: " + ds9); // Deep Space Nine
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Arrays: forEach()

You can loop over the items of an array by executing the `Array.prototype.forEach()` method with a callback function that takes up to 3 parameters: `item`, `index`, and `array`. The callback function is called once for each item of the array in ascending order.

ES2015

```
let hotels = ['Vulcan Inn', 'Star Fleet Motel', 'Romulus Resorts', 'Klingon BnB'];
// You can loop over the array like this
hotels.forEach(function(item, index, array) {
   console.log(index + " = " + item);
});
// Or like this, skipping the last two parameters
hotels.forEach(function(item) {
   console.log(item);
});
// Or using the ES2015 arrow syntax
hotels.forEach(item => console.log(item));
```

https://repl.it/NHqn/1

Arrays: map()

You can iterate over the items of an array by executing the `Array.prototype.map()` method with a callback function that takes an item as input parameter and returns some value. The callback function is called once for each item of the array in ascending order. The returned values are pushed into a newly created array. The original array remains unchanged.

ES2015

```
let hotels = [{name: 'Vulcan Inn', stars: 2},
              {name: 'Star Fleet Motel', stars: 3},
              {name: 'Romulus Resorts', stars: 5},
              {name: 'Klingon BnB', stars: 1}];
let hotelLabels = hotels.map(item => item.name + " (" +
item.stars + '\u2606' + ")");
console.log(hotels); // original array is unchanged
console.log(hotelLabels);
// ["Vulcan Inn (2☆)", "Star Fleet Motel (3☆)", "Romulus
// Resorts (5☆)", "Klingon BnB (1☆)"]
```

https://repl.it/NHqw/1

Arrays: reduce()

You can reduce the items of an array to some single value by executing the `Array.prototype.reduce()` method with a callback function which is executed once per array item.

ES2015

```
let hotels = [{name: 'Vulcan Inn', stars: 2},
              {name: 'Star Fleet Motel', stars: 3},
              {name: 'Romulus Resorts', stars: 5},
              {name: 'Klingon BnB', stars: 1}];
// Reduce the array to one of the highest rated hotels
let maxStars = (previousValue, currentValue) => {
   if (currentValue.stars > previousValue.stars) {
       return currentValue;
   } else {
       return previousValue;
   }
};
let highestRatedHotel = hotels.reduce(maxStars);
console.log(highestRatedHotel); // {name:"Romulus Resorts",stars:5}
```

https://repl.it/NHrB/3

Arrays: map() and reduce()

Technische Universität Berlin

You can combine the `map()` and `reduce()` methods as shown in the following example.

ES2015

```
let hotels = [{name: 'Vulcan Inn', stars: 2},
             {name: 'Star Fleet Motel', stars: 3},
             {name: 'Romulus Resorts', stars: 5},
             {name: 'Klingon BnB', stars: 1}];
// Reduce the array to one of the highest rated hotels
let maxStars = (previousValue, currentValue) => {
   if (currentValue > previousValue) {
       return currentValue;
   } else {
       return previousValue;
   }
};
let highestRating = hotels.map(hotel => hotel.stars).reduce(maxStars);
console.log(highestRating); // 5
```

https://repl.it/NIBx/1

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Arrays: sort()

The `Array.prototype.sort()` method sorts the items of your array *in place*, i.e., changes the order of items in your original array. You can supply a callback function to the `sort()` method to implement your own comparator logic.

ES2015

```
let hotels = [{name: 'Vulcan Inn', stars: 2},
              {name: 'Star Fleet Motel', stars: 3},
              {name: 'Romulus Resorts', stars: 5},
              {name: 'Klingon BnB', stars: 1}];
function sortByStars(previousValue, currentValue) {
   return previousValue.stars - currentValue.stars;
}
hotels.sort(sortByStars);
console.log(JSON.stringify(hotels));
// [{"name":"Klingon BnB","stars":1},{"name":"Vulcan Inn",
// "stars":2},{"name":"Star Fleet Motel",
// "stars":3},{"name":"Romulus Resorts","stars":5}]
```

https://repl.it/NICS/1

Arrays: find()

Execute the `Array.prototype.find()` method with a callback function that implements your search criteria. If the callback function returns true, the corresponding array item is returned by the `find()` method.

ES2015

```
let hotels = [{name: 'Vulcan Inn', stars: 2},
             {name: 'Star Fleet Motel', stars: 3},
             {name: 'Romulus Resorts', stars: 5},
             {name: 'Klingon BnB', stars: 1}];


function findHotel(hotel) {
   return hotel.name === 'Vulcan Inn';
}

let found = hotels.find(findHotel);
console.log(found); // {name: "Vulcan Inn", stars: 2}
```

https://repl.it/NICf/3

Arrays: filter()

Execute the `Array.prototype.filter()` method with a callback function that specifies one or more filter criteria, such as the minimum hotel rating, as demonstrated with the `aboveAverage()` callback function below. Thereby, you create a new array which is assembled of items that match the filter criteria.

ES2015

```
let hotels = [{name: 'Vulcan Inn', stars: 2},
             {name: 'Star Fleet Motel', stars: 3},
             {name: 'Romulus Resorts', stars: 5},
             {name: 'Klingon BnB', stars: 1}];
function aboveAverage(hotel) {
   return hotel.stars >= 3;
}
let aboveAverageHotels = hotels.filter(aboveAverage);
console.log(JSON.stringify(aboveAverageHotels));
// [{"name":"Star Fleet Motel","stars":3},
// {"name":"Romulus Resorts","stars":5}]
```

https://repl.it/NICq/1

What did we learn today?

- Variable scoping in JavaScript ES5 and ES2015
- Functional programming with JavaScript
  - Anonymous function expressions, ES2015 arrow function notation
  - Nested functions
  - Closures
  - Currying
- Array and Array extras API

PROGRAMMING JAVASCRIPT II

Learning objectives

- Object-oriented programming
    - What is an object in JavaScript?
    - How can you create objects?
    - How does prototype-based inheritance work?
    - Learn the basic ES2015 and TypeScript syntax for object-oriented programming
    - How can we modularize application using ES2015 module syntax?
- Asynchronous & concurrent programming
    - What is the event-loop and concurrency model in JavaScript?
    - What are callback functions and ES2015 `Promises`?

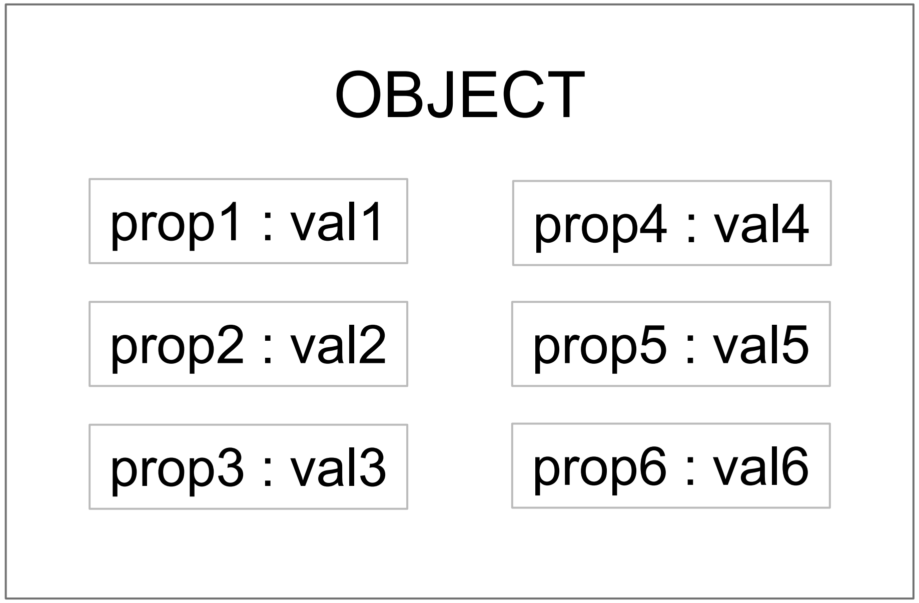
OBJECT-ORIENTED JAVASCRIPT

Object-Oriented JavaScript

- JavaScript is an object-oriented (OO) language
  - Objects encapsulate data and provide clearly defined interfaces by which they can be used
  - OO programming advocates a way of structuring your code that promotes software re-use, data privacy, and other desirable objectives that help to make the code that you write more robust and maintainable.
- JavaScript (ES5) is quite different from other OO languages, such as Java
  - Prototype-based inheritance
  - No classes, no interfaces, no types, no private/public modifiers
  - Functions are objects

Objects

- You can imagine a JavaScript object as a bag that contains a set of **properties**.
- A property has a name and a value.
  - The property name must be a string.
  - The property value can be of any type (string, number, object, etc.).
- You can think of a JavaScript object as a string-to-value map (“dictionary”, “hashtable”).



Methods

- **Method** = a function is assigned as value to the property of an object
- A method is not different from other functions in JavaScript, except that it is invoked through the invocation context of the object.
- The context of the object can be accessed from within the function body via the `this` keyword.

Example

HOTEL

properties of the "Hotel" object

name : `'VulcanInn'`	numRooms : `12`
location : `'Mars'`	stars : `3`

log : `function () { console.log(this); }`

method

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Creating Objects

- JavaScript offers different ways to create objects:
  - Object literal
  - `Object.create()`
  - new constructor

Creating Objects: Object Literal Pattern

JavaScript

```javascript
var hotel = {
 chain: 'Vulcan Inn',
 location: 'Berlin',
 numRooms: 12,
 stars: 3,
 guests: ['McCoy', 'Spock'],
};
```

https://repl.it/NIDH/1

Creating Objects:
Object Constructor Function

JavaScript

```
function Hotel(hotelChain, location, numRooms) {
 this.chain = hotelChain;
 this.location = location;
 this.numRooms = numRooms;
};

var hotel = new Hotel('Vulcan Inn', 'Mars', 100);
```

https://repl.it/NIDH/1

Prototype-based Inheritance

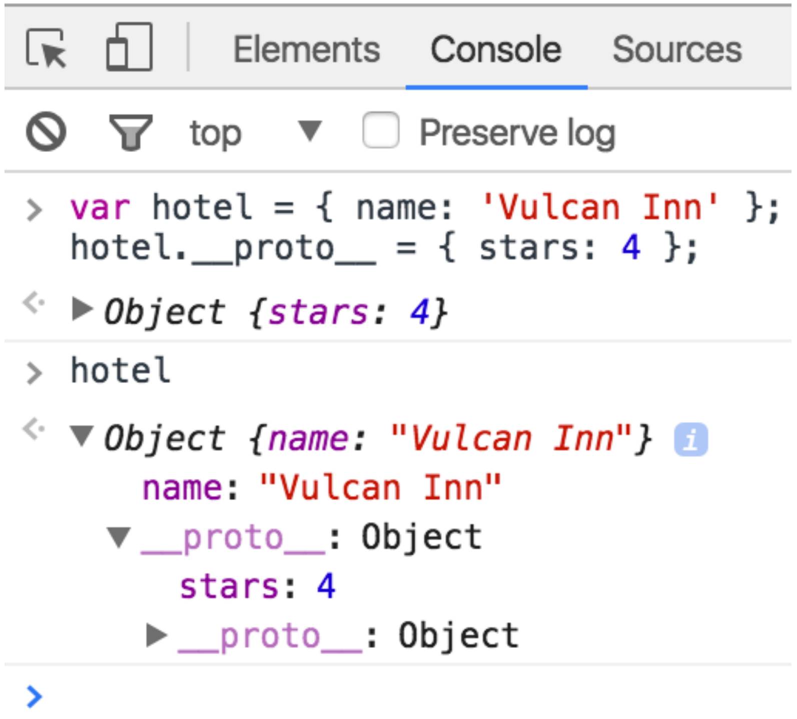
- Inheritance is a core concept of object-oriented programming that allows developers to create more specialized (child) classes from more generic (parent) classes.
- JavaScript, however, does not have a concept of classes. Instead, JavaScript uses **prototype-based inheritance**:
  - Every object has a (parent) prototype object that passes down its properties to the child object.
  - The prototype serves as a “blueprint” when you create a new objects.

- This is realized by providing each object with a `[[Prototype]]` property (equivalent to the `__proto__` property) that references another object. The referenced object has its own prototype property that might link to yet another object. This leads to a **prototype chain**. The prototype chain ends at a (parent) object without a prototype, *i.e.,* where the object’ prototype property is null.

Example

JavaScript

```
var hotel = { name: 'Vulcan Inn' };
hotel.__proto__ = { stars: 4 };
```



```
> var hotel = { name: 'Vulcan Inn' };
  hotel.__proto__ = { stars: 4 };
<· ▶ Object {stars: 4}
> hotel
<· ▼ Object {name: "Vulcan Inn"}
      name: "Vulcan Inn"
    ▼ __proto__: Object
        stars: 4
      ▶ __proto__: Object
>
```

Object.create()

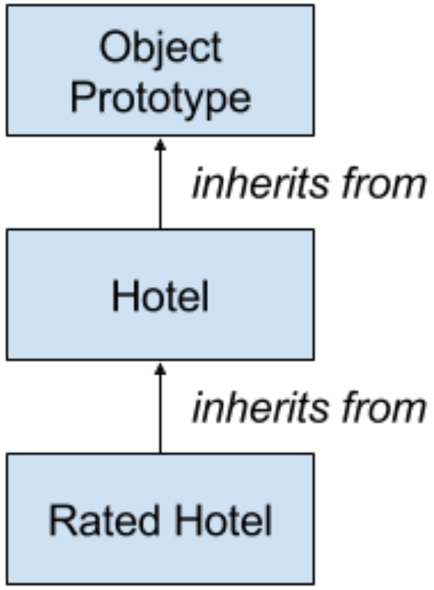
- Directly assigning objects to the `__proto__` property is a bad practice. Instead, you should use the `Object.create()` constructor pattern.

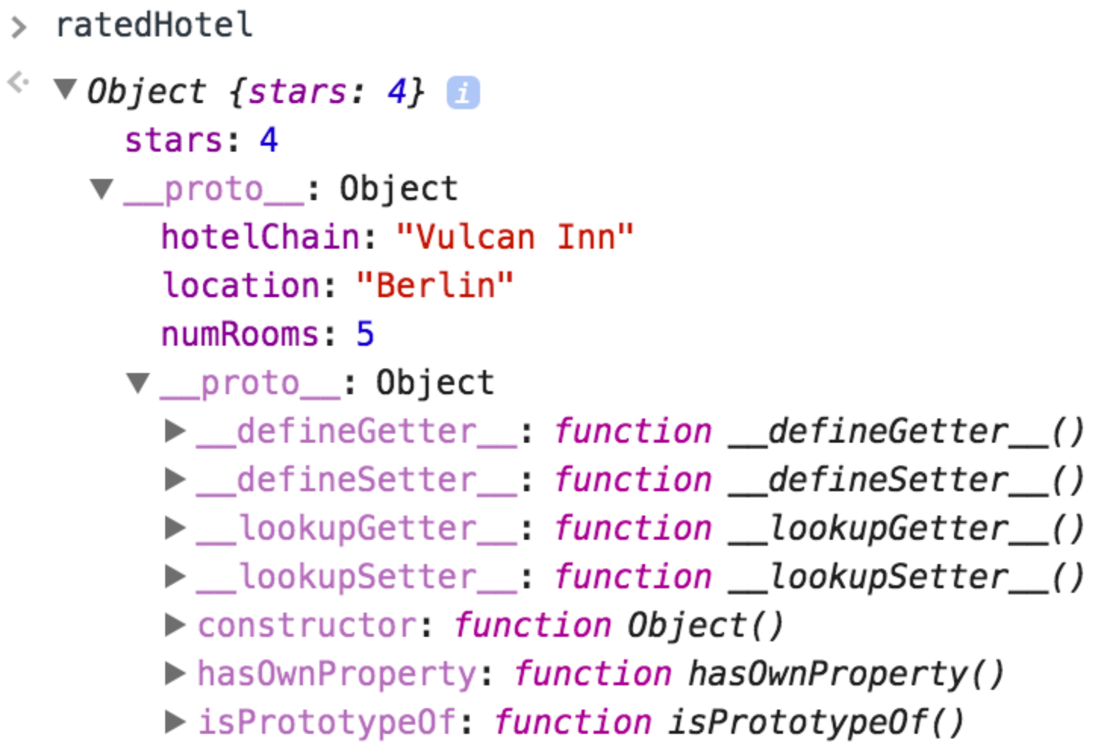
JavaScript

```javascript
// Object literal constructor pattern
var hotel = {
   hotelChain: 'Vulcan Inn',
   location: 'Berlin',
   numRooms: 5,
 };
// Object.create() constructor pattern
var ratedHotel = Object.create(hotel);
ratedHotel.stars = 4;
console.log(ratedHotel.location); // Berlin
console.log(Object.prototype.isPrototypeOf(hotel)); // true
console.log(hotel.isPrototypeOf(ratedHotel)); // true
console.log(ratedHotel.isPrototypeOf(hotel)); // false
```

https://repl.it/NIEc/1

Prototype chain

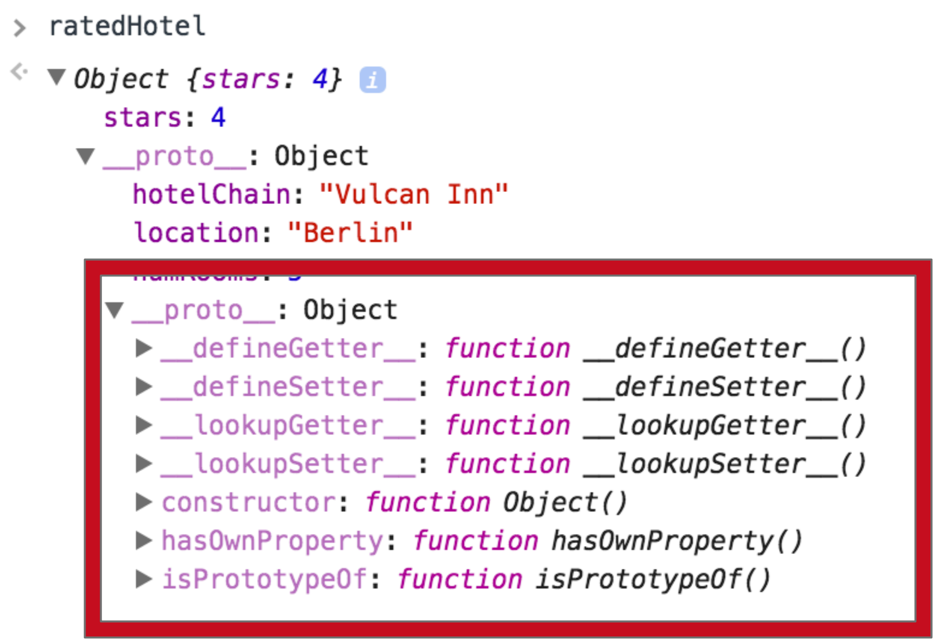




```
> ratedHotel
<· ▼ Object {stars: 4} i
      stars: 4
    ▼ __proto__: Object
        hotelChain: "Vulcan Inn"
        location: "Berlin"
        numRooms: 5
      ▼ __proto__: Object
        ▶ __defineGetter__: function __defineGetter__()
        ▶ __defineSetter__: function __defineSetter__()
        ▶ __lookupGetter__: function __lookupGetter__()
        ▶ __lookupSetter__: function __lookupSetter__()
        ▶ constructor: function Object()
        ▶ hasOwnProperty: function hasOwnProperty()
        ▶ isPrototypeOf: function isPrototypeOf()
```

The global “Object Prototype”

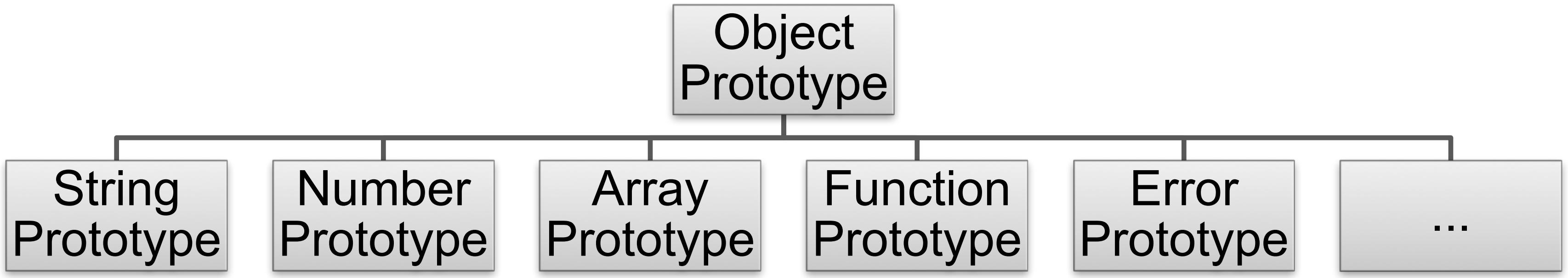
- By default, all JavaScript objects inherit their properties from a single global `Object` prototype object. All JavaScript objects inherit built-in properties and methods, for example:
  - `toString()` to return a string representation of an object, or
  - `hasPrototypeOf()` to test if the calling object has a certain object (passed as argument) in its prototype chain.



```
> ratedHotel
<· ▼Object {stars: 4}
     stars: 4
   ▼__proto__: Object
      hotelChain: "Vulcan Inn"
      location: "Berlin"
    ▼__proto__: Object
      ▶__defineGetter__: function __defineGetter__()
      ▶__defineSetter__: function __defineSetter__()
      ▶__lookupGetter__: function __lookupGetter__()
      ▶__lookupSetter__: function __lookupSetter__()
      ▶constructor: function Object()
      ▶hasOwnProperty: function hasOwnProperty()
      ▶isPrototypeOf: function isPrototypeOf()
```

Specialized object prototypes

- In fact, all standard built-in JavaScript objects inherit from their own specialized prototype objects (which inherit from the Object prototype).
- For example, if you create an array object instance, the array inherits from the `Array` prototype which supplies your array with useful methods, such as `pop()`, `push()`, etc.



Constructor function

JavaScript

```javascript
function Hotel(hotelChain, location, numRooms) {
 this.chain = hotelChain;
 this.location = location;
 this.numRooms = numRooms;
 this.log = function () { console.log(this); };
};

var h = new Hotel('Vulcan Inn', 'Mars', 100);
h.stars = 4;
h.log();

// Hotel { chain: 'Vulcan Inn', location: 'Mars',
// numRooms: 100, log: [Function], stars: 4 }
```

https://repl.it/NIFu/4

Constructor function + prototype property

JavaScript

```
function Hotel(hotelChain, location, numRooms) {
 this.chain = hotelChain;
 this.location = location;
 this.numRooms = numRooms;
};

Hotel.prototype = {
   log: function () {
       console.log(this);
     },
 };

var h = new Hotel('Vulcan Inn', 'Mars', 100);
h.stars = 4;
h.log();
// { chain: 'Vulcan Inn', location: 'Mars', numRooms: 100, stars: 4 }
```

https://repl.it/NIFu/4

Constructor function + prototype property

Alternatively, we could add the log**()** method to the existing (inherited) prototype object instead of “overwriting” the entire prototype object, like this:

JavaScript
`Hotel.prototype.log = function () { console.log(this); };`

https://repl.it/NIFu/4

Constructor function + prototype property

If you use the `new` keyword to invoke the constructor, the constructor function's prototype property is assigned to the newly created object's `[[Prototype]]` (aka `__proto__`) property.

JavaScript

```
var o = new Foo();
```

JavaScript

```
var o = new Object();
o.__proto__ = Foo.prototype;
Foo.call(o);
```

Function prototype

- The `Function` prototype has a few interesting methods that allow you to pass a custom **`this`** value into a function:
  - `apply()`
  - `call()`
  - `bind()`

Indirect function invocation with `call()`

JavaScript

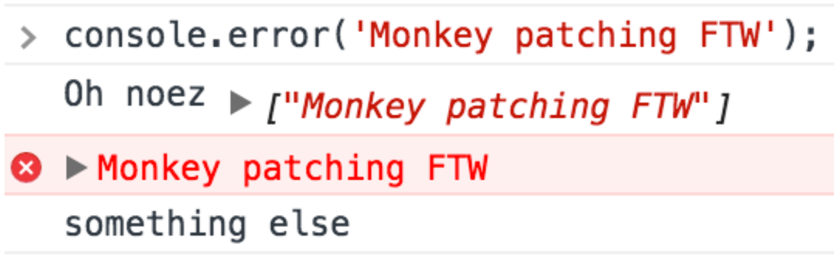
```
var hotel = {
 name: 'Vulcan Inn',
};

function praise(adj) {
 console.log(this.name + ' is ' + adj);
}

praise('fantastic'); //  is fantastic
praise.call(hotel, 'fantastic'); // Vulcan Inn is fantastic
```

https://repl.it/NIHL/6

Indirect function invocation with apply()

JavaScript

```javascript
// Monkey patching example using apply()
var originalConsoleErr = console.error;
console.error = function () {
 // We do something else here
 console.log('Oh noez', arguments);
 // Call the original function
 originalConsoleErr.apply(this, arguments);
 // We can do something else here
 console.log('something else');
};
```



```
> console.error('Monkey patching FTW');
  Oh noez ▶ ["Monkey patching FTW"]
⊗ ▶ Monkey patching FTW
  something else
```

https://repl.it/NIHL/6

Bound Function (BF) creation with bind()

JavaScript

```
name = 'global scope';

var hotel = {
 name: 'Vulcan Inn',
 praise: function (adj) {
   console.log(this.name + ' is ' + adj);
 },
};

var praiseCopy = hotel.praise;
var praiseBindCopy = hotel.praise.bind(hotel);

hotel.praise('fantastic'); // Vulcan Inn is fantastic
praiseCopy('weird'); // global scope is weird
praiseBindCopy('pretty good'); // Vulcan Inn is pretty good
```

https://repl.it/NIHL/6

ES2015 Classes

ES2015 classes are “syntactic sugar” on top of the prototype-based inheritance of JavaScript. Classes are intended to provide a clearer declarative syntax for implementing object-oriented software design patterns.

ES2015

```
class Hotel {
 constructor(name, location) {
   this.name = name;
   this.location = location;
 }
 description() {
   return this.name + ' ' + this.location;
 }
}
let h = new Hotel('Vulcan Inn', 'Moon');
console.log(h.description()); // Vulcan Inn Moon
```

ES2015 Classes: extends, super

ES2015

```
class RatedHotel extends Hotel {
 constructor(name, location, rating) {
   super(name, location);
   this.rating = rating;
 }

 description() {
   return super.description() + ' (' + this.rating + ')';
 }
}

let rh = new RatedHotel('Klingon BnB', 'Earth', 2);
console.log(rh.description());
// Klingon BnB Earth (2)
```

ES2015 Classes: static

ES2015

```
class Hotel {
 constructor(name, location) {
   this.name = name;
   this.location = location;
 }
 static frame(str) {
   let hashtags = '';
   for (let i = 0; i < str.length; i++) {
     hashtags += '#';
   }
   return hashtags + '\n' + str + '\n' + hashtags;
 }
 description() {
   return Hotel.frame(this.name + ' ' + this.location);
 }
}
let h = new Hotel('Vulcan Inn', 'Moon');
console.log(h.description());
// ###############
// Vulcan Inn Moon
// ###############
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

TypeScript Classes

TypeScript	JavaScript

```
class Hotel {
   private _name: string;
   public location: string;
   constructor(name: string,
       location: string) {
       this._name = name;
       this.location = location;
   }
   describe() {
       return this._name + ' ' +
           this.location;
   }
}

let hotel = new Hotel('Elon
Resorts', 'Mars');
console.log(hotel.describe());
console.log(hotel.location);
```

~~console.log(hotel._name);~~

```
var Hotel = (function () {
   function Hotel(name, location) {
       this._name = name;
       this.location = location;
   }
   Hotel.prototype.describe =
       function () {
           return this._name + ' '
           + this.location;
   };
   return Hotel;
}());

var hotel = new Hotel('Elon
Resorts', 'Mars');
console.log(hotel.describe());
console.log(hotel.location);
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

TypeScript Classes

- As you can see, the TypeScript `Hotel` class transpiles to a closure: an anonymous function definition expression that is instantly invoked, *i.e.*, `(function () { … }());`
  - The anonymous function wraps around a nested function declaration statement with function name `Hotel` and two parameters `name` and `location`.
  - The `describe()` method is added to the `Hotel.prototype` object.
  - As last statement the `Hotel` function is returned.
  - By invoking the anonymous (outer) function, the `Hotel` function is returned and assigned to the `Hotel` variable.

TypeScript Classes

- Furthermore, TypeScript introduces **`private`**, **`public`**, and **`protected`** modifiers that work similar as their Java equivalents.
- By convention, private variables are prefixed with an underscore (this is just a convention and not a requirement).
- The private variable `hotel._name` should not be directly accessed whereas the `hotel.location` property and `hotel.describe()` method are public and can be accessed (all variables without modifiers are by default public).
- The last line in the TypeScript code (`console.log(hotel._name);`) will throw an error when you try to transpile the code to JavaScript.

TypeScript Abstract Classes

- Abstract classes are base classes that cannot be instantiated, but can be subclassed (other classes can inherit abstract classes).
- An abstract class can contain properties and methods just like a normal class.
- In addition, an abstract class may contain abstract methods which declare a method signature without an implementation.

TypeScript

```
abstract class AbstractHotel {
   protected _name: string;
   constructor(public name: string) {
       this._name = name;
   }
   abstract stats(): string;
   abstract addGuest(guest: string): void;
}
```

TypeScript Abstract Classes (2)

TypeScript

```
class Hotel extends AbstractHotel {
   private _guests: string[];
   constructor(public name: string, public guests: string[]) {
       super(name);
       this._guests = guests;
   }
   stats(): string {
       return this._name + ' has ' + this._guests.length + ' guests.';
   }
   addGuest(guest: string): void {
       this._guests.push(guest);
   }
   removeGuests(): void {
       this._guests = [];
   }
}
```

TypeScript Abstract Classes

TypeScript

```
let hotel: AbstractHotel;
//hotel = new AbstractHotel();
// error: Cannot create an instance of the abstract class
// 'AbstractHotel'.
hotel = new Hotel('Starfleet BnB', ['Worf']);
hotel.addGuest('Picard');
console.log(hotel.stats());
//hotel.removeGuests();
// error: Property 'removeGuests' does not exist on type
// 'AbstractHotel'.
```

Duck Typing & TypeScript Interfaces

- Duck-typing is a technique to determine the identity of an object by its behavior and not its "class" (i.e., prototype object)
  - "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck" (James W. Riley)
  - Duck-typing is one of TypeScript's core principles: "that type-checking focuses on the shape that values have".

- TypeScript adds an `interface` keyword to the JavaScript syntax.
  - An interface give the "duck type" a name.
  - With interfaces, you can define contracts that specify how your code should be used.

TypeScript Interfaces

TypeScript

```
interface HotelBooking {
   name: string;
   nights: number;
}

function bookHotel(bkg: HotelBooking) {
   console.log(bkg.name + ' has booked ' + bkg.nights + ' nights.');
}

let b = { name: 'Picard', nights: 3 };
bookHotel(b); // Picard has booked 3 nights.

// let b2 = { name: 'Worf' };
// bookHotel(b2);
// Argument of type '{ name: string; }' is not assignable to parameter
// of type 'HotelBooking'.
//  Property 'nights' is missing in type '{ name: string; }'.
```

TypeScript Interfaces

- The duck-typing approach of TypeScript let's us specify an `interface` that the argument which is passed to the `bookHotel` function must comply with.
- Note that we do not need to write boilerplate code, like stating that the object which we pass as an argument must implement the interface.
- Since TypeScript just like plain JavaScript takes a duck-typing approach, it is sufficient that the shape of the object matches the shape of the interface. The booking `{ name: 'Picard', nights: 3 }` fulfills the contract which we specify with the interface. The second booking `{ name: 'Worf' }` won't transpile to JavaScript because the `nights` property is missing, and therefore the shape of our object does not fulfill the contract as specified by our `interface`.

MODULAR JAVASCRIPT

Modular Software Design

- *Modularity* of software programs can be achieved by breaking down large programs into multiple smaller programs, each of which fulfills a single responsibility.
  - Modules generally improve maintainability because they are self-contained. If you change the implementation of a module, you should not expect that this change affects other parts of the code which are outside of the module.
  - Modules improve software reuse because you can use the same module in different projects. If the module requires an update, the update can more easily propagate to all projects that use the module.

ES2015 Modules

- ES2015 introduces a module syntax. The ES2015 module syntax allows you to export and import modules. Any (variable, function, class, …) declaration can be exported with the `export` keyword.
- There are 2 kinds of export statements:
  - named exports (multiple exports per module), and
  - default exports (1 export per module).

ES2015 Modules: Named exports

lib1.js

```
// ES2015 named module exports
export const sqrt = Math.sqrt;
export function square(x) {
 return x * x;
}
```

main.js

```
// ES2015 module imports
import { sqrt, square } from 'lib1.js';
let x = 2;
let y = 4;
console.log(sqrt(y) + ' is the square root of ' + y);
console.log(square(x) + ' is the square of ' + x);
```

ES2015 Modules: Default exports

lib2.js

```
// ES2015 default module exports
import { sqrt, square } from 'lib1.js';

export default class {
 constructor(x, y) {
   this.x = x;
   this.y = y;
 }

 static distance(a, b) {
   return sqrt(square(a.x - b.x) + square(a.y - b.y));
 }
}
```

ES2015 Modules: Default exports (2)

main.js

```
// ES2015 module imports
import Point from 'lib2.js';

let a = new Point(1, 1);
let b = new Point(2, 2);
console.log('The euclidian distance between a and b is ' +
Point.distance(a, b));
```

ES2015 Modules: Re-export

lib3.js

```
// ES2015 module imports and exports
export * from 'lib1.js';
import Point from 'lib2.js';

export default function (p1, p2) {
 let dist = Point.distance(p1, p2);
 return `Dist. from (${p1.x}, ${p1.y}) to (${p2.x}, ${p2.y}) 
is ${dist}`;
}
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

ES2015 Modules: Re-export (2)

main.js

```
// ES2015 module imports
import Point from 'lib2.js';
import distanceText, { square } from 'lib3.js';

let a = new Point(2, 2);
let b = new Point(square(a.x), square(a.y));
console.log(distanceText(a, b));
```

ES2015 Modules

- ES2015 offers language-level support for a **module syntax**. Language design decisions of ES2015 modules aim at a syntax and structure that allow static analysis and optimization, as well as support for cyclic dependencies.

- However, a system for module loading is still work in progress and not part of the ES2015 specification. A **module loader** is needed to locate and resolve dependencies of a module at runtime before that module can be executed.
  - In a previous ES2015 draft, the module loader should support both synchronous and asynchronous module loading.
  - Since module loading is not part of ES2015, you need to use a third-party module loader library, such as System.js if you want to use ES2015 modules.

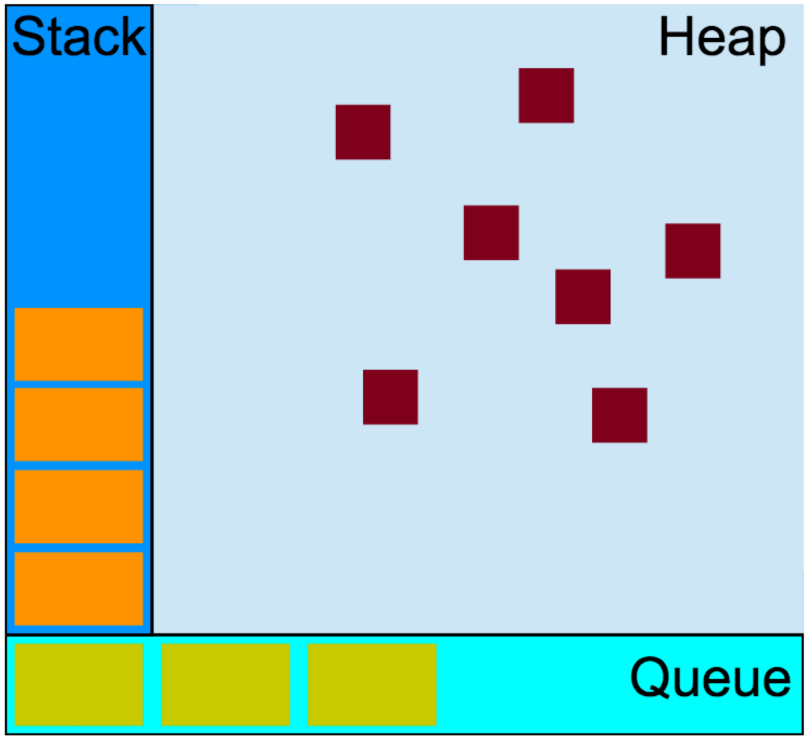
Technische Universität Berlin

ASYNCHRONOUS & CONCURRENT JAVASCRIPT

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

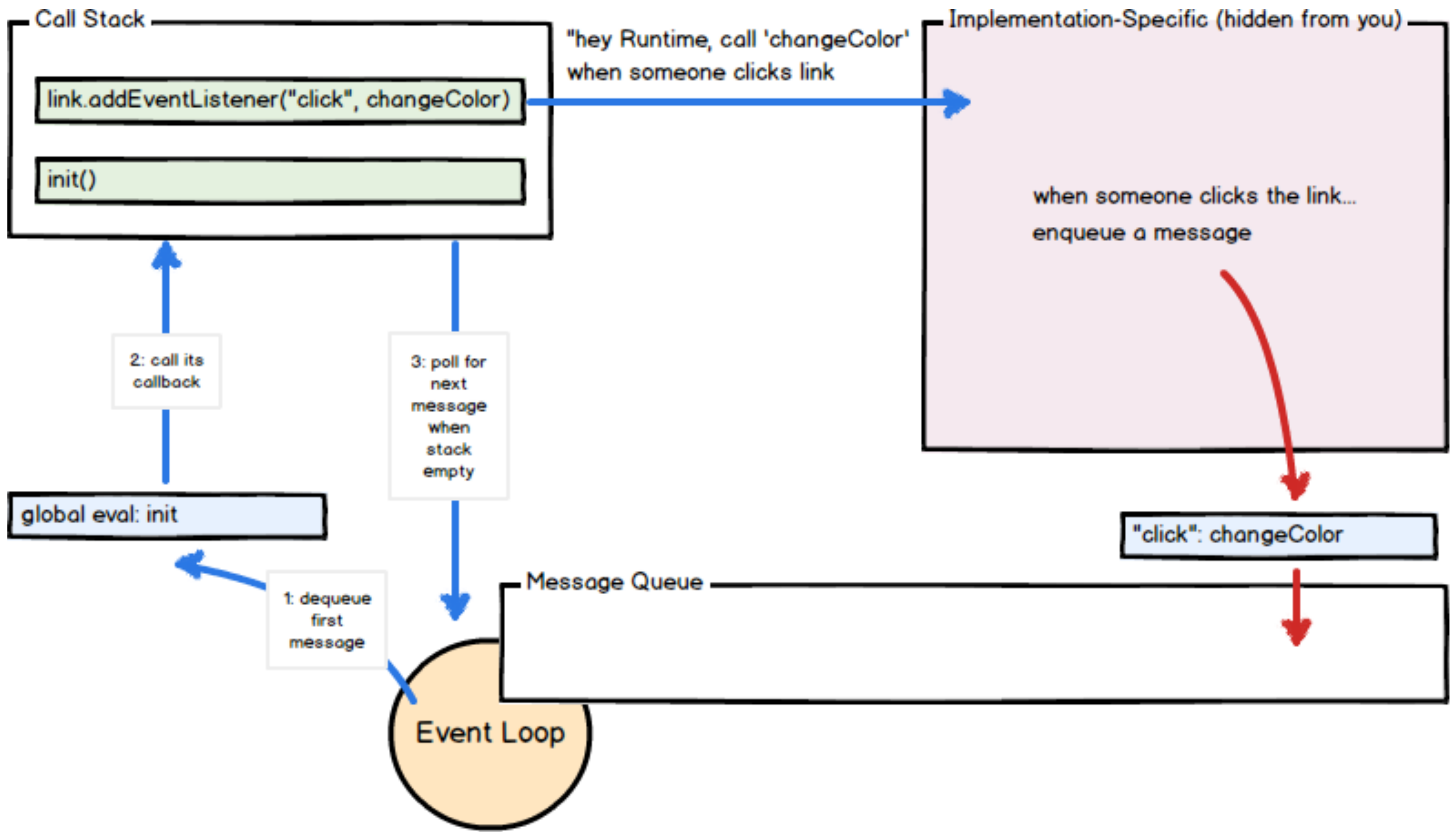
Concurrency Model & Event Loop

- On a theoretical level, the JavaScript runtime consists of three main structures:
    1. a **heap** where JavaScript objects allocate memory,
    2. a **stack** where frames are stacked one over the other, and
    3. a message **queue** which contains a list of messages.



https://developer.mozilla.org/en/docs/Web/JavaScript/EventLoop

Concurrency Model & Event Loop


Call Stack
link.addEventListener("click", changeColor)
init()
"hey Runtime, call 'changeColor' when someone clicks link
Implementation-Specific (hidden from you)
when someone clicks the link... enqueue a message
2: call its callback
3: poll for next message when stack empty
global eval: init
"click": changeColor
Message Queue
1: dequeue first message
Event Loop

http://blog.carbonfive.com/2013/10/27/the-javascript-event-loop-explained/

Callback Functions

- All access to I/O should be non-blocking, for example, accessing the file system, a remote database, or a web service API. I/O is characterized by
  - Nondeterministic execution time
  - Relatively long execution time (compared to memory access)

- The use of callback functions is a design pattern for implementing non-blocking programs.
  - You can pass a (callback) function as an argument to another function.
  - The latter function can then invoke the callback function within its function body.

Callback Functions: Example

JavaScript

```
function doSomeIO() {
 console.log('Assume we do some expensive IO here.');
}

doSomeIO();
console.log('Yay! I/O is done.');

// Assume we do some expensive IO here.
// Yay! I/O is done.
```

Callback Functions: Example (2)

JavaScript

```javascript
function doSomeIO() {
 // simulate a long-lasting 100ms operation
 setTimeout(function () {
   console.log('Assume we do some expensive IO here.');
 }, 100);
}

doSomeIO();
console.log('Yay! I/O is done.');

// Yay! I/O is done.
// Assume we do some expensive IO here.
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Callback Functions: Example (3)

JavaScript

```
function mylog(text) {
 console.log(text);
}

function doSomeIO(callback) {
 setTimeout(function () {
   console.log('Assume we do some expensive IO here.');
   callback('Yay! I/O is done.');
 }, 100);
}

doSomeIO(mylog);

// Assume we do some expensive IO here.
// Yay! I/O is done.
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Callback Functions: Example (4)

JavaScript

```
function mylog(error, result) {
 if (error) {
   return console.error('An error occured', error);
 }
 console.log(result);
}
function doSomeIO(callback) {
 setTimeout(function () {
   console.log('Assume we do some expensive IO here.');
   var err = null;
   var failed = true;
   if (failed) {
     err = new Error('I/O failed');
   }

   callback(err, 'Yay! I/O is done.');
 }, 100);
}
doSomeIO(mylog);
// Assume we do some expensive IO here.
// An error occurred Error: I/O failed ...
```

Callback Hell

- The use of many dependent callbacks can lead to a design anti-pattern which is commonly known as callback hell.
- The anti-pattern emerges when you try to write JavaScript code that uses dependent callback functions in an imperative top-down manner, thereby forming a 90° turned “code pyramid” of nested callback function.

Callback Hell: Example

JavaScript

```javascript
function welcome(callback) {
 callback(null, 'Welcome');
}
welcome(function (err, res) {
   if (err) {
     console.error('Oh no!', err); return;
   }
   var toPrint = (function (err, res) {
     if (err) {
       console.error('Wut?', err); return;
     }
     return ' to' + (function (err, res) {
       if (err) {
         console.error('Srsly?', err); return;
       }
       return ' hell!!';
     })();
   })();
   console.log(res + toPrint);
 });
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

Callback Hell: Example (2)

JavaScript

```javascript
function welcome(callback) {
 callback(null, 'Welcome');
}
function toParadise(err) {
 if (err) {
   console.error('Wut?', err); return;
 }
 return ' to' + paradise();
}
function paradise(err) {
 if (err) {
   console.error('Srsly?', err); return;
 }
 return ' paradise \u2601 \u2601';
}
welcome(function (err, res) {
   if (err) {
     console.error('Oh no!', err); return;
   }
   var toPrint = toParadise();

   console.log(res + toPrint);
 });
```

Technische Universität Berlin

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

ES2015 Promises

- ES2015 introduces the concept of a `Promise` object that “promises” to perform an operation in the future.
- With Promises, JavaScript developers can avoid spaghetti code of nested callback functions (callback hell).
- Writing more readable and better structured asynchronous code generally improves maintainability and reduces the risk of making mistakes and introducing bugs.

ES2015 Promises

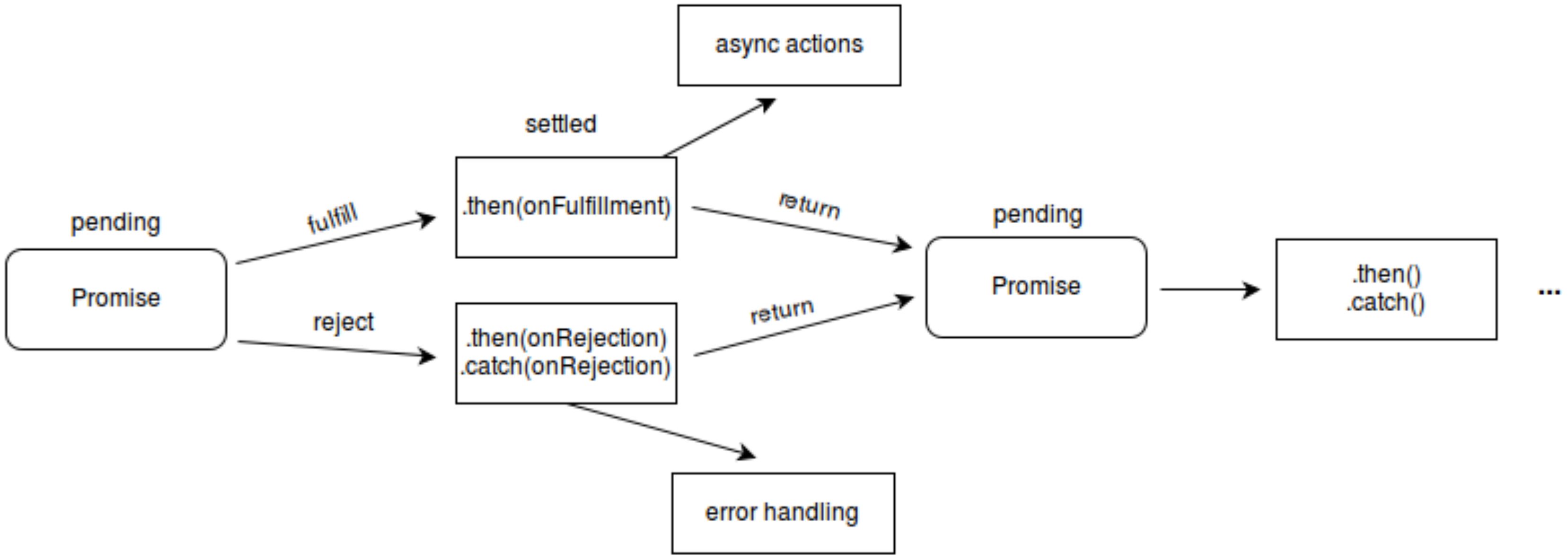


Image Source https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

ES2015 Promises: Example

ES2015

```
// Long operation
let promise = new Promise((resolve, reject) => {
 let start = Date.now();
 let howLong = Math.random() * 2000;
 setTimeout(() => {
   let dur = Date.now() - start;
   resolve('Long operation finished after ' + dur + 'ms.');
 }, howLong);
});

promise.then(data => console.log(data));
console.log('This is a quick operation.');
```

```
This is a quick operation.
Long operation finished after 613ms.
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

ES2015 Promises: Example (2)

ES2015

```
// Long operation with potential error
let promise = new Promise((resolve, reject) => {
 let start = Date.now();
 let howLong = Math.random() * 2000;
 if (howLong > 1000) {
   reject('This is taking too long: ' + howLong + 'ms.');
 }

 setTimeout(() => {
   let dur = Date.now() - start;
   resolve('Long operation finished after ' + dur + 'ms.');
 }, howLong);
});

promise.then(data => console.log(data)).catch(error =>
console.log(error));
```

```
This is taking too long: 1362.87360628215ms.
```

https://repl.it/NIVV/2

ES2015 Promises: Example (3)

ES2015

```
// Call the Wiki API without a Promise
const request = require('request');
const url = require('url');

let data = {
 action: 'opensearch',
 format: 'json',
 search: 'Planet Jupiter',
};

let wikiUrl = 'https://en.wikipedia.org/w/api.php' + url.format({
query: data });

request(wikiUrl, (error, response, body) => {
 if (!error && response.statusCode == 200) {
   console.log(body); // Show the HTML for the Google homepage.
 }
});
```

```
["Planet Jupiter",["Planet
Jupiter"],[""],["https://en.wikipedia.org/wiki/Planet_Jupiter"]]
```

ES2015 Promises: Example (4)

ES2015

```
// Call the Wiki API with a Promise
const request = require('request');
const url = require('url');
let data = {
 action: 'opensearch',
 format: 'json',
 search: 'Planet Jupiter',
};
let wikiUrl = 'https://en.wikipedia.org/w/api.php' + url.format({ query: data });
let promise = new Promise((resolve, reject) => {
 request(wikiUrl, (error, response, body) => {
   if (!error && response.statusCode == 200) {
     resolve(body);
   } else {
     reject(error);
   }
 });
});
promise.then(data => console.log(data)).catch(error => console.log(error));
```

```
["Planet Jupiter",["Planet
Jupiter"],[""],["https://en.wikipedia.org/wiki/Planet_Jupiter"]]
```

ES2015 Promise Chain

ES2015

```
let double = (input) => {
   return new Promise((resolve, reject) => {
       setTimeout(() => {
         console.log(input);
         resolve(input * 2);
       }, 500);
     });
 };

// Chain of Promises
double(1).then(double).then(double);
```

```
1
2
4
```

https://repl.it/NIVn/1

ES2015 Promise Chain

ES2015

```
let double = (input) => {
  return new Promise((resolve, reject) => {
      setTimeout(() => {
        console.log(input);
        resolve(input * 2);
      }, 500);
    });
};

// Chain of Promises
double(1).then(double).then(double);
```

Closure (function expression)

Nested function (that uses the Promise constructor)

Promise-style callback function

```
1
2
4
```

ES2015 Promise Chain

The return value of a then operation is a `Promise`. Therefore, you can even chain up functions with little boilerplate code, as shown in the following example:

ES2015

```
let p1 = new Promise((resolve, reject) => resolve('a'));
p1.then(val => val + 'b').then(val => console.log(val));
```

```
ab
```

ES2015 Promise.all()

Technische Universität Berlin

ES2015

```
let p1 = new Promise((resolve, reject) => resolve('p1'));
let p2 = new Promise((resolve, reject) => resolve('and p2'));
let p3 = new Promise((resolve, reject) => {
 setTimeout(() => {
   resolve('.... and p3');
 }, 100);
});

// Wait for the fulfillment of all promises
Promise.all([p1, p3, p2]).then(values => {
 values.forEach(val => console.log(val));
});
```

```
p1
.... and p3
and p2
```

ISEngineering
Wirtschaftsinformatik –
Information Systems Engineering

What did we learn today?

- Object-oriented programming
  - Objects are “bags of properties” (“string-to-value mapping”)
  - You can create objects with either object literal syntax, `Object.create()` method, or constructor function + **`new`** keyword
  - In JavaScript, inheritance is based on the concept of object prototypes
    - The prototype (parent) object can be accessed through the `[[Prototype]]` (aka `__proto__`) property
    - The prototype object of a (child) object can have its own prototype, thereby forming a prototype chain
  - ES2015 offers a class syntax (which however, is just sugar syntax over prototype-based inheritance) and TypeScript offers additional syntax for abstract classes, interfaces, etc.
  - ES2015 introduces a module syntax (however lacks a module loader)

What did we learn today?

- Asynchronous & concurrent programming
  - JavaScript programs execute in a single event-loop thread.
  - Therefore it is important that they are asynchronous (non-blocking).
    - Callback functions are a design pattern for asynchronous programming (not part of the JavaScript language specification). This pattern can be used because JavaScript functions are objects and can be passed to other functions as arguments.
    - `Promises` are a concurrent programming concept and part of the ES2015 syntax that make JavaScript programs which use asynchronous functions better maintainable by avoiding callback hell.

Outlook

The next lecture:

- Server-side software development with JavaScript