Bioinformatics Resources

Lecture & Exercises
Prof. B. Rost, Dr. L. Richter, J. Reeb
Institut für Informatik I12
Slides by D. Nechaev
Origins

- Brendan Eich, Netscape, 1995
- Language for the Netscape Navigator browser
- Called JavaScript because marketing
- Inspired by Scheme (functional language) and Self (prototypes), not by Java
JScript

- Microsoft reverse engineered JavaScript
- Called it JScript and added to the Internet Explorer in 1996
- JScript & JavaScript had incompatibilities, browser war started
ECMAScript

- ECMAScript – standardized specification, 1997
- latest edition (ECMAScript 6) finalized in June 2015
- ES6 is still not completely supported by the browsers
- This talk presents ES5 features
Language for the Browser

- Designed as a browser language with no input/output capabilities
- Interaction with the environment (HTML page)
- Use `script` tag to add js files to the page

```html
<script src="filename" type="text/javascript"></script>
```
Node.js

- Platform for running JavaScript on the server side (2009)
- Utilizes Google’s V8 JavaScript engine to interpret JavaScript code
- Allows to write both backend and frontend code in JavaScript
JavaScript

- Complex applications started being developed in the second half of 2000s as Web 2.0 emerged
- Currently is one of the most popular languages
- Allows to create rich User Interfaces for the Web (plots, diagrams, classical UI elements, etc.)
Simple Execution Environment

- One of the simplest ways to run JavaScript code is to use Chrome Google Developer Tools
- Cmd+Alt+I or Ctrl+Alt+I
- Alternatively, right click, choose `Inspect`, select `Console` tab
First look at variables

- `var` keyword allows to declare and initialize variables (several at once)
- valid identifiers begin with a letter, `$`, or `_`
- other symbols can be letters, digits, `$`, and `_`

```javascript
< var identifier;
< var factory, $obj, _private;
< var meaning = 42, level = 9000;
```
Data Types
Data Types

- **Primitive:**
  - Number
  - String
  - Boolean
  - null
  - undefined

- **Referenced:**
  - Object
  - Array
  - Date
  - Regexp
  - Function
Number, 0.1 + 0.2

- Only one type for numbers - float
- uses 8 bytes for number representation
- be careful with financial data

< 0.1 + 0.2 === 0.3;
> false
< 0.1 + 0.2;
> 0.30000000000000004
Math.pow

- Standard arithmetical operations and modulo are defined for numbers
- `Math` object implements more complex math operations

```java
< 10 % 3;
> 1
< Math.pow(2, 10);
> 1024
```
Number literals

- Numbers written in the source code (as in the examples above), decimal by default, hexadecimal begin with `0x`
- Float literals contain `.` or `e`
- Prefix operator `~` is used for negative numbers

< 0xF;
> 15
< 101e-1;
> 10.1
< -10;
> -10
Special Values

- `Infinity` & `-Infinity` are returned in the case of overflow
- NaN is returned for the operations with no defined value

< -1e1000;
> -Infinity
< 1 / 0;
> NaN
Wrapping object

- Numbers are a primitive type (by value)
- `Number` constructor allows to create wrapping objects

```
< typeof 7;
> "number"
< typeof new Number(7);
> "object"
< (new Number(7)).valueOf()
> 7
```
Strings

- String is a sequence of 16-bit Unicode symbols
- No char type
- String literal begins and ends with `\`\` or `""` (no difference)

```
< 'Rostlab\t\n' === "Rostlab\t\n";
> true
```
Complex Strings

- No interpolation, concatenate
- Break strings into multiple lines with `\`

```< 'Rostlab ' + 2016;
> "Rostlab 2016"
< 'a\n< b';
> "ab"```
Immutability & wrapping object

- Strings are a primitive immutable type
- Methods like `replace()` and `toLowerCase()` return a new string
- `String` constructor allows to create wrapping objects

```
< typeof new String("Rostlab");
> "object"
< (new String("Rostlab")).valueOf();
> "Rostlab"
```
Boolean

- Boolean type has two values - `true` and `false`
- Any expression used in a conditional is automatically converted to Boolean
- `null`, `undefined`, 0, `NaN`, and `""` => `false`
- anything else => `true`
Wrapping object

- `Boolean` constructor allows to create wrapping objects
- Be careful – wrapping object for `false` is evaluated as true!

```javascript
< if (new Boolean(false)) {
  console.log("is true");
}
> is true
```
undefined

- Special type `undefined` has only one value – `undefined`
- Uninitialized variable, nonexistent array element or object property
- Returned by functions with no explicit return value
- Function parameter with no corresponding argument

```javascript
< var x;
< x
> undefined
```
**undefined**

- global variable `undefined` returns value `undefined`, which is the only value of type `undefined`
null

- A bit complicated
- `null` is a keyword
- `typeof` returns `object`
- However, `null` has no properties and no methods unlike other objects

```javascript
< typeof null;
> "object"
```
null

- Just a few of inbuilt functions return null
- Thus, it is considered a good practice to use `null` instead of `undefined` in user code to indicate absence of value
Objects

- JS Object is an associative array (think python dictionary), i.e. a set of properties, where each property is a key-value pair
- Objects are stored by reference

```javascript
var a = { name: 'Rostlab' }, b = a;
b.name = 'Rostlab 2016';
a.name;
"Rostlab 2016"
```
Object Contents

- No classes, so no limits on the names and values of the properties
- An object can have arbitrary properties
- An object’s property’s value can be another object
- To access an object’s property use dot notation or square bracket notation

```javascript
var lct = { name: 'Dr. Richter' };
var cls = { name: 'BioinfRes', lecturer: lct };
cls.lecturer.name;
> "Dr. Richter"
cls['lecturer'].name;
> "Dr. Richter"
cls.lecturer['name'];
> "Dr. Richter"
```
No classes?

- Yes, no classes
- Prototypal inheritance is used for code reuse and modeling groups of objects with similar behavior
- Arrays, functions, regular expressions, and dates are objects and they get their behavior via prototypal inheritance
- We’ll look at it in details later
Array

- An object that stores an ordered group of values
- Dynamic (can change size) and untyped (can store values of different types)
- Other symbols can be letters, digits, `$`, and `_`

```javascript
var array = ['Rostlab'];
array.push(2016);
array;
> ['Rostlab', 2016]
```
Any value

- An array can hold any value including itself

```javascript
var array = ['Rostlab'];
array.push(array);
array[1][0];
"Rostlab"
```
Date

Objects created with the `Date` constructor have methods for handling dates and time

```javascript
var now = new Date();
now.getFullYear();
> 2016
```
Regexp objects describe the string template

```
var regexp = new RegExp('^Rostlab');
regexp.test('Rostlab 2016');     // true;
regexp.test('2016 Rostlab');     // false
```
Functions

- Functions are so called ‘first class citizens’:
  - can store in variables, arrays, objects’ properties
  - can pass to other functions as arguments
  - can return from other functions
- Functions are multiary (don’t know what `multiary` means? Compare with `unary`, `binary`, etc. ;-) )
- If we don’t pass an argument for a certain parameter, its value is `undefined`
- Can access all arguments passed to a function with the `arguments` keyword
Functions

- Grasping functions is essential for JavaScript
- We’ll look at functions in detail

```javascript
< function printArgs(a, b, c) {
    console.log(a);
    console.log(b);
    console.log(c);
}
< printArgs('Rostlab', 2016);
> Rostlab
> 2016
> undefined
```
Summary

- `Boolean`, `Number`, and `String` are primitive immutable objects stored by value.
- We can create wrapper objects for primitive values.
- That’s exactly what happens when we call a method on a primitive value – a wrapper object is created, a method is called on the wrapper object, then the wrapper object is destroyed.

```
< (2016).toString();
> "2016"
< ' Rostlab '.trim();
> "Rostlab"
```
Summary Pt. 2

- `null` & `undefined` are the only values of their respective data types
- No wrapper objects for them
Summary Pt. 3

- Objects are stored by reference
- Objects are mutable
Type conversions

- Functions `Number`, `Boolean`, and `String`
- Functions `parseFloat` and `parseInt`
- Methods `toString`, `toExponential`, `toFixed`, and `toPrecision`

```
< String(2016);
> "2016"
< Boolean(2016);
> true
< (123.45).toExponential();
> "1.2345e+2"
```
Strong typing

- Some languages (e.g., Python) have strong typing
- Strong typing means we can’t mix values of different types

```python
>>> 'Rostlab ' + 2016
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```
Weak typing

- JavaScript is a dynamic weakly typed language
- JavaScript doesn’t care!

< 'Rostlab ' + 2016;
> "Rostlab 2016"
Weak typing

• **REALLY** doesn’t care

```
< '7' * '4';
> 28
```
Weak typing

- Please, be very careful
- Rules of automatic type casting are complex

```c
< '6' - '4';
> 2
< '6' + '4';
> "64"
```
Double equality operator

- Double equality operator automatically does type conversion
- That makes it intransitive

```c
< '0' == 0 && 0 == ' ';  // true
< '0' == '';  // false
```
Triple equality operator

- Triple equality operator returns `true` only if operands belong to the same data type and hold the same value.

- It is considered a good practice to use `===` and `! ==` operators instead of double equality.

```javascript
< '0' === 0;  // false
> false
< 0 === '';   // false
> false
```
Magic strings

- Automatic type casting is dangerous but allows to perform some fun tricks
Magic strings

- Automatic type casting is dangerous but allows to perform some fun tricks

```
< +!![]+(+!![])++[]+(+![])+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[]+(+!![])++[-]>>

> "2016 rostlab"
```
Control Statements
Control Statements

- `if – else`, `switch` and ternary operator
- `while` and `for` loop
- `break` and `continue` instructions
If else

- Can use `if` again after `else`
- Use curly braces if want to execute more than one statement
- It is considered a good practice to always use curly braces

```javascript
if (hand === 'Rock') {
  console.log('Rock beats scissors, you win');
} else if (hand === 'Paper') {
  console.log('Scissors beat paper, you lose');
} else {
  console.log('Try again');
}
```
Ternary operator

- Ternary operator allows to write conditional in a more compact way
- `if – else` conditional: if `condition do1 else do2`
- Ternary operator: condition ? do1 : do2

```< console.log(grade > 4 ? 'Fail' : 'Pass')>```
Ternary operator

- We can use ternary operator inside another ternary operator
- Just because we can doesn’t mean we should
- What is the following code doing?

```c
< return (a<b) ? (b<c) ? b : (a<c) ? c : a : (a<c) ? a : (b<c) ? c : b;
```
Ternary operator

- The following code returns median of $a$, $b$, and $c$
- Unreadable and unmaintainable

```c
< return (a<b) ? (b<c) ? b : (a<c) ? c : a : (a<c) ? a : (b<c) ? c : b;
```
Switch

- `switch` allows to choose one of many statements

- We can do the same with multiple if–else, but `switch` allows to write more compact and readable code

```c
switch (month) {
    case 0:
        'Jan';
        break;
    case 1:
        'Feb';
        break;
    case 11:
        'Dec';
        break;
    default:
        'Unknown';
        break;
}
```
Switch

- `switch` compares value of the conditional with the value of each `case` until `===` operator returns true

- Once equality is found, respective code block is executed

- Unless we `break` out of `switch` statement, all of the below code blocks will be executed as well

- `default` code block is executed if no equality is found

```javascript
< switch(month) { 
  case 0:
    'Jan';
    break;
  case 1:
    'Feb';
    break;
  ... 
  case 11:
    'Dec';
    break;
  default:
    'Unknown';
    break;
}
```
For loop

- JavaScript `for` loop is analogous to C `for` loop
- `for (init; condition; post-iteration-statement)`
  loop-statement

```
< for (var i = 0; i < 10; i++)
  console.log(i);
> 0
> 1
> ...
> 9
```
Cthulhu loop

- We can easily create an infinite loop (;;)
- But just because we can doesn’t mean we should

```javascript
< for (;;) // in the name of Cthulhu
  console.log('All your CPU are belong to Cthulhu!');
```
For–in loop

- `for-in` loop iterates over object properties’ names
- To get property’s value use square bracket notation

```javascript
var cls = {name: 'Rostlab', year: '2016', lecturer: 'Dr. Richter'};
for (var p in cls) {
  console.log(p + ': ' + cls[p]);
}
```

> name: Rostlab
> year: 2016
> lecturer: Dr. Richter
While

- `while` loop executes a block code while the condition is true
- while (condition) statement

```javascript
< var i = 0;
< while (i < 10) {
    console.log(i);
    i++;
}
```

> 0
> 1
> ...
> 9
Do while

- `do – while` loop executes a block code, then checks the condition and starts another iteration if it is true
- Will run at least once

```javascript
var i = 11;
do {
    console.log(i);
    i++;
} while (i < 10);
> 11
```
Break, continue, label

- `break` aborts current loop and `switch` statement
- `continue` starts next loop iteration
- We can use labels with `break` and `continue`
- Labels look suspiciously similar to GOTO
- Research and use at your own risk

```javascript
< outerLoop:
  while (true) {
    innerLoop:
      while (true) {
        console.log('No more loops');
        break outerLoop;
      }
    console.log('You won\'t see me!');
  }
> No more loops
```
Logical operators

- We can use logical operators in complex conditions
- Use `&&` for `AND`, `||` for `OR`, and `!` for `NOT`

```c
< true && false;
> false
< true || false;
> true
< !false;
> true
```
Changing priority of operations

- Use parenthesis to change priority of operations

\[
(2 + 2) \times 2 = 8
\]

\[
2 + 2 \times 2 = 6
\]
Functions
Function Declaration

- Use keyword `function` followed by a function’s name and a list of parameters

```< function functionName(par1, ..., parN) {
  statements
}
```
Function Expression

- Allows to create anonymous functions
- We can assign created function to a variable

```javascript
< var varName = function(par1, ..., parN) {
    statements
}
```
Function Expression with a Name

- We can even use a function expression with a name
- In such case we will only be able to call a function by it’s name inside the function itself

```javascript
< var myFactorial = function factorial(n) {
  if (n > 0) {
    return n * factorial(n - 1);
  } else {
    return 1;
  }
};
< factorial(3);
> Uncaught ReferenceError: factorial is not defined(…)
< myFactorial(3);
> 6
```
Declaration and Expression

- Both define functions
- Function declaration requires a name, while function expression allows anonymous functions
- Thanks to function declaration *hoisting* we can use functions before they are declared, as if function declaration was lifted to the top of the block
- Finally, we can’t declare functions inside loops, conditionals and other control statements, while function expressions have no such limits
Function Declaration Hoisting

```javascript
function factorial(n) {
    result = 1;
    for (var i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

> 120
```
Working with Arguments

- Function *has* parameters and *takes* arguments
- We can invoke a function with a different amount of arguments than parameters
- Parameters with no corresponding argument will have value `undefined`

```javascript
function myFunc(str1, str2) {
    console.log(str2);
}
myFunc('only one argument!');
```

> undefined
Working with Arguments

- If the number of arguments is larger than the number of parameters, “excessive” arguments simply will not be used.
- We still can access them thanks to a special variable `arguments`.
- `arguments` object contains all the arguments passed to a function.

```javascript
< function myFunc(str1, str2) {
  console.log(arguments[3]);
};
myFunc('one', 'two', 'three', 'four');
> four
```
Working with `arguments` object

- Although `arguments` object has `length` property and allows to access its elements by index, it is not an array and does not have array methods

```javascript
> ['one', 'two', 'three', 'four'].indexOf('three');
2
> function myFunc(str1) {
    return arguments.indexOf('three');
}
myFunc('one', 'two', 'three', 'four');
> TypeError: arguments.index0f is not a function
```
Working with `arguments` object

- Still, we can convert `arguments` object to an array using `Array.prototype.slice.call(arguments)`
- We’ll investigate this code in details a little bit later, for now treat it as a snippet

```javascript
< ['one', 'two', 'three', 'four'].indexOf('three');
> 2
< function myFunc(str1) {
  var argumentsArray = Array.prototype.slice.call(arguments);
  return argumentsArray.indexOf('three');
}
myFunc('one', 'two', 'three', 'four');
> 2
```
Function Return Value

- Function always returns a value
- By default `undefined` is returned
- To return a different value use `return` keyword

```javascript
< function square(a) {
  return Math.pow(a, 2);
}

square(7);
> 49
```
Automatic Semicolon Insertion

- Remember about ASI and put the returned value on the same line as `return` keyword

```javascript
< function f1() {
    return {
        year: 2016
    }
}
< function f2() {
    return {
        year: 2016
    }
}
< f2()
> Object {year: 2016}
< func1()
> undefined
Automatic Semicolon Insertion

- Thanks to the Automatic Semicolon Insertion code on the left is equal to code on the right

```< function f1()
  {
    return
    {
      year: 2016
    }
  }
```

```< function f1()
  {
    return; //←ALARM
    {
      year: 2016
    }
  }
```
No Block Scoping in JavaScript

- In some other languages, declaring a variable inside a block creates a variable only inside that block
- In JavaScript a variable will be visible outside this block as well

```javascript
for (var i = 0; i < 3; i++) {
    console.log(i);
}
```

> 0
> 1
> 2
> 3 // if you wonder, why it’s 3, study loops again
Functions Create Scope

- Functions are the only way to control scope
- Each function creates a new scope
- A variable declared in a scope is accessible inside all of the inner scopes

```
< var name = 'Rostlab';
    function printName() {
        console.log(name);
    }
    printName();
> Rostlab
```
Inner Functions and Outer Scope

- Let’s say it in a different way – we can access inside a function all of the variables declared in this function’s outer scope
- We can both read and write such variables

```javascript
< var name = 'Rostlab';
    function rebranding() {
        name = 'Bioinformatics Chair';
    }
    rebranding();
    name
> "Bioinformatics Chair"
```
Variable shadowing

- Declaring a variable with the same name as a variable in an outer scope blocks access to the outer variable.

```
< var name = 'Rostlab';
    function rebranding() {
        var name = 'Bioinformatics Chair';
    }
    rebranding();
    name

> "Rostlab"
```
Variable hoisting

- Variable declarations are hoisted as well as function declarations
- A variable declaration is ‘lifted’ to the top of its scope

```javascript
< var name = 'Rostlab';
  function rebranding() {
    name = 'Bioinformatics Chair';
    var name;
  }
  rebranding();
  name
> "Rostlab"
```
Immediately Invoked Function Expressions allow us to prevent loitering

Variables declared inside IIFE are inaccessible outside

```javascript
(function() {
  var iDoNotPolluteOuterScope;
}());

iDoNotPolluteOuterScope

Uncaught ReferenceError: iDoNotPolluteOuterScope is not defined
Higher order functions

- A higher order function takes a function as an argument or returns a function as a return value

```javascript
< function add(x) {
    return function innerAdd(y) {
        return x + y;
    }
}
add(4)(5)
> 9
```
`add(x)` takes an integer as an argument and returns another function that takes another integer as an argument.

`add(4)` returns `innerAdd(y)` and we immediately call it passing 5 as an argument.

```javascript
< function add(x) {
    return function innerAdd(y) {
        return x + y;
    }
}

add(4)(5)
> 9
```
Closures

● How can `innerAdd` function access variable `x` after `add` function has been invoked?
● It’s all about the closures
● JavaScript inner function has an access to outer variables even when an outer function has been run
● A function “encloses” its environment
● A closure is function plus its context
Classic example – a simple counter

- IIFE + closure

```javascript
< var counter = (function() {
    var cntr = 0;
    return function() {
        return ++cntr;
    }
})();

> counter();
> 1
> counter();
> 2
> counter();
> 3
```
Objects
Objects – Recap

- An object is an associative array, i.e. a set of properties, of pairs `name - value`
- An object is mutable and is passed by reference, it can have arbitrary properties whose values can be of any type, including functions and other objects
- Arrays, functions, regular expressions and dates are objects as well
Creating an Object

- `new Object()` creates a new object with no properties
- Having created an object, we can add properties one by one

```javascript
var cls = new Object();
cls.name = 'Bioinformatics Resources';
cls.year = 2016;
cls
```

> Object {name: "Bioinformatics Resources", year: 2016}
Object literal

- An object literal allows to create an object with its properties filled
- An object literal is a list of properties (name-value pairs) in curly braces

```
< var cls = {
    name: 'Bioinformatics Resources',
    year: 2016,
};

cls

> Object {name: "Bioinformatics Resources",
    year: 2016}
```
Property names

- We can use valid JS identifiers as property names and access such properties via dot notation.
- A valid identifier starts with a letter, an underscore (_), or a dollar sign ($).
- Subsequent characters can be letters, digits, underscores, or dollar signs.

```javascript
< var validNames = {
    valid: 'valid name',
    _$$: 'valid as well',
    _$$_100500: 'this is valid too',
};
validNames._$$_100500
> "this is valid too"
```
Property names

- We can use strings that are not valid identifiers as well
- We have to quote them and access via square bracket notation

```javascript
var validNames = {
  '100500': 'this is valid',
};
validNames['100500'] > "this is valid"
```
Property values

- A property value can be of any type
- Functions that are properties of an object are called ‘methods’
- Inside of a method `this` keyword refers to the object the method is called on

```javascript
var cls = {
    name: 'Bioinformatics Resources',
    year: 2016,
    print: function() {
        console.log(this.name + ', ' + this.year);
    }
}
cls.print();
```

> Bioinformatics Resources, 2016
Working with properties

- We can read, write, add, and delete properties

```javascript
< var cls = {
  name: 'Bioinformatics Resources',
  year: 2016,
}
cls.chair = 'Rostlab';
console.log(cls);
> Object {name: "Bioinformatics Resources", year: 2016,
  chair: "Rostlab"}
< delete cls.chair;
console.log(cls)
> Object {name: "Bioinformatics Resources", year: 2016}
< cls.chair
> undefined
```
Iterating over properties

- `for-in` loop iterates over object properties’ names
- To get property’s value use square bracket notation

```javascript
var cls = {
    name: 'Bioinformatics Resources',
    year: 2016,
    print: function() {
        console.log(this.name + ', ' + this.year);
    }
};
for (var p in cls) {
    console.log(p + ': ' + cls[p]);
};
> name: Bioinformatics Resources
  year: 2016
  print: function () {
      console.log(this.name + ', ' + this.year);
  }
```
Checking property existence

- `hasOwnProperty()` method allows us to check if an object has a given property

```javascript
< cls.hasOwnProperty('year')
> true
< cls.hasOwnProperty('month')
> false
```
Comparing objects

- An object is equal only to itself
- In other words, equality operator returns `true` only if both variables hold a reference to the same object

```javascript
< lecture = cls; lecture === cls; > true
< {name: 'value'} === {name: 'value'} > false
```
Prototypes

- By the way, we never added `hasOwnProperty` method to our objects, but we still were able to use it. Why?

- Because prototypes!

```javascript
< ({}).hasOwnProperty('year');
> false
```
Prototypal Inheritance

- Each object has a special connection to another object, to a prototype. When we try to access a property that is absent in an object, we access a property of the prototype. We can say that object inherits its prototype properties.

- `Object.prototype` is a prototype of all objects created via an object literal. `Object.prototype` has `hasOwnProperty` method, and that’s exactly what we invoked.
Prototypal Inheritance

- If we add a property to a prototype, all of the objects inheriting from this prototype will get immediate access to the new property.

```javascript
Object.prototype.easterEgg = 'Hello, world!';
cls.easterEgg
> "Hello, world!"
```
Prototypal Inheritance

- What happens if we add `easterEgg` property to our `cls` object?

```javascript
< cls.easterEgg = function() {
    console.log('Have I messed up my prototype?');
}
cls.easterEgg();
> Have I messed up my prototype?
< Object.prototype.easterEgg;
> "Hello, world!"
< ({}).easterEgg;
> "Hello, world!"
```
Functions Revisited
Functions are Objects

- A function is an object
- `name` property’s value is function’s name (an empty string for an anonymous function)
- `length` property’s value is an amount of parameters

```javascript
< (function() {}).name
> ""
< var foo = function bar(a, b, c) {}
  foo.name;
> "bar"
< foo.length;
3
```
Function.prototype

- Function object is connected to a prototype as well
- `Function.prototype` is the prototype of functions

```javascript
< Object.getPrototypeOf(foo) === Function.prototype
> true
```
`apply` & `call`

- Functions inherit `apply` and `call` methods

```
var print = function() {
    console.log(this.name + ', ' + this.year);
};
var cls = {
    name: 'Bioinformatics Resources',
    year: 2016,
}
print.apply(cls);
> Bioinformatics Resources, 2016
print.call(cls);
> Bioinformatics Resources, 2016
```
`apply` & `call` methods allow to call a function as if it was a method of the object that is passed as a first argument.

- `this` keyword returns passed object.

```javascript
var returnThis = function() {
    return this;
}
returnThis.apply(cls) === cls; > true
```
`apply` & `call`

- Arguments passed after the object become arguments of the function
- `call` takes these arguments simply as arguments, `apply` takes an array

```javascript
< var print = function(times) {
  for (var i = 0; i < times; i++) {
    console.log(this.name);
  }
};

print.call(cls, 3);
```
Array as arguments with `apply`

- Using `apply` we can unpack arguments, that is, turn an array into arguments for a function.
- A mnemonic allows to remember which of two methods takes an array: “apply” and “array” start and end with the same letters.

```javascript
< Math.max(3, 5, 9);
> 9
< Math.max([3, 5, 9]);
> NaN
< Math.max.apply(null, [3, 5, 9]);
> 9
```
Arguments Array

- Previously shown way of turning `arguments` object into an actual array should be clear now

```
< Array.prototype.slice.call(arguments);
```
Calling One Object’s Methods on Another Object

- We can invoke one object’s methods on another object using `apply` and `call` or simply by adding methods to another object.

```javascript
var cls = {
  name: 'Bioinformatics Resources',
  year: 2016,
  print: function() {
    console.log(this.name + ', ' + this.year);
  }
};
var anotherCls = {
  name: 'Protein Prediction',
  year: 2016,
};
anotherCls.print = cls.print;
anotherCls.print();
> Protein Prediction, 2016
```
Late Binding

- `this` becomes bound to an object at the invocation time
- Late binding allows to use `apply` & `call` or copy methods from one object to another
- Unless...
Function.prototype.bind

- `Function.prototype.bind` returns a new function bound to a specified object

- `this` keyword will always return bound object

```javascript
< var cls = {
    name: 'Bioinformatics Resources',
    year: 2016,
};
cls.print = (function() {
    console.log(this.name + ', ' + this.year);
}).bind(cls);
var anotherCls = {
    name: 'Protein Prediction',
    year: 2016,
};
anotherCls.print = cls.print;
anotherCls.print();
> Bioinformatics Resources, 2016
```
`this`

- What will `this` return if a function hasn’t been bound to an object and is not invoked as a method or via `apply` or `call`?
- By default `this` returns the global object
- In the browser it’s `window` object

```javascript
< function checkThisOut() {
  console.log(this);
};
checkThisOut();
```
In an inner function `this` by default returns the global object

```javascript
var cls = {
    name: 'Bioinformatics Resources',
    testInnerFunc: function() {
        return (function() {
            console.log(this);
        })();
    }
};
cls.testInnerFunc();
```
`this` and that

- However, we already know how to deal with it
- We can either bind an inner function or cache `this` in another variable, for example `that`

```javascript
< var cls = {
    name: 'Bioinformatics Resources',
    testInnerFunc: function() {
        var that = this;
        return (function() {
            console.log(that);
        }());
    }
};
cls.testInnerFunc();
> Object {name: "Bioinformatics Resources"}
```
Objects Revisited
Objects with Similar Behaviour

- We want to have objects with similar behaviour
- We can add explicitly add methods to each object
- We can add methods to `Object.prototype`
- Is there a better solution?
Constructor Functions

- Constructor functions allow to create a group of objects such that each object belonging to a group has similar behaviour

```javascript
< function Swordsman(name, hp, damage) {
    this.name = name;
    this.hp = hp;
    this.attacks = 2;
    this.damage = damage;
}
var swordsman = new Swordsman('Arthur', 100, 40);
swordsman
> Swordsman {name: "Arthur", hp: 100, attacks: 2, damage: 40}
```
Constructor Functions

- `new` keyword creates a new object
- Value of the `prototype` property of the constructor function becomes the prototype of the new object
- `this` returns the new object inside the constructor function

```javascript
< Object.getPrototypeOf(swordsman) === Swordsman.prototype true
```
Constructor Functions

- Each function, including constructor functions, has `prototype` property
- Value of `prototype` property is an object
- This object becomes the prototype of every object created by a corresponding constructor function

```javascript
< Swordsman.prototype.getTotalDamage = function() {
    return this.attacks * this.damage;
}
swordsman.getTotalDamage()
> 80
```
Adding Properties to a Prototype

- We can add properties of any type and not just functions

```javascript
< Swordsman.prototype.house = 'Lannister';
  swordsman.house
> "Lannister"
```
No Access Control

- We now know enough to write a simple videogame
- Inspired, we sat down to work on it
- And suddenly we realize that all of the object’s properties are public, are visible from the outside
- JavaScript has no way of declaring certain properties as private or protected, they are all public
- Our swordsman’s damage and attack stats are exposed to the outside world
Closures as a Solution

- We can solve this problem with closures

```javascript
function Swordsman(name, hp, damage) {
    this.name = name;
    this.hp = hp;
    var dmg = damage;
    var attacks = 2;
    this.getTotalDamage = function() {
        return attacks * dmg;
    };
}
var swordsman = new Swordsman('Arthur', 100, 40);
swordsman.getTotalDamage();
> 80
swordsman.dmg
> undefined
swordsman.damage
> undefined
```
Closures as a Solution

- `dmg` and `attacks` variables are defined in the constructor
- These variables are accessible inside `getTotalDamage` function
- Thanks to closures these variables exist even after constructor invocation has been completed
- It is impossible to access these variables from the outside
- We can make properties and methods "private" this way
Access Control VS Prototypes

- Closures allow us to limit access to certain properties
- Everything has a price, however
- We can’t access such properties inside the prototype methods
- Thus, we have to choose between access control with closures and code reuse with prototype chains
Prototype Chain?

- Yes, a chain
- We can create an object via a constructor function
- Make this object a prototype of another object

```javascript
< function Lannister() {}
Lannister.prototype.house = 'Lannister';
function Swordsman() {}
Swordsman.prototype = new Lannister;
function Archer() {}
Archer.prototype = Swordsman.prototype;
var swordsman = new Swordsman();
swordsman.house
> "Lannister"
```
Prototypal Inheritance

- When we invoke a specific method on an object
- First, JavaScript looks for a method with a given name in the object
- Then, in that object’s prototype
- Then, in that object’s prototype’s prototype
- ...
- Finally, in `Object.prototype`
Dangers of Constructors

- Constructor function is a regular function (no special keywords, no naming restrictions, etc.), it’s easy to mix a constructor function and a regular function
- Error: calling a constructor function without `new` (`this` will not be bound to a new object)
- Error: calling a regular function that explicitly returns an object with `new` (it will still return that object)
Dangers of Constructors

- Error: calling a constructor function without `new` (`this` is bound to the global object)

```javascript
< function Constructor() {
    this.name = 'Bioinformatics Resources';
};
var cls = Constructor();
cls
> undefined
< name
> "Bioinformatics Resources"
```
Dangers of Constructors

- Error: calling a regular function that explicitly returns an object with `new` (it will still return that object)

```javascript
function Constructor() {
    return {
        name: 'Bioinformatics Resources'
    };
}
var cls = new Constructor();
Object.getPrototypeOf(cls) === Constructor.prototype;  // false
Object.getPrototypeOf(cls) === Object.prototype;    // true
```
Wrapping Up

- 5 different ways to call a function with respect to `this`:
  - Function invocation
  - Method invocation
  - Constructor invocation
  - Indirect invocation with `apply` and `call`
  - Bound invocation with `bind`
Wrapping up

- 4 ways to create an object:
  - Default `Object` constructor
  - Object literal
  - User-defined constructor
  - Closures
Constructor vs Closure

- Closures allow to control access
- Prototypes allow to reuse code and use less memory by storing properties and methods in the prototype
Object.create

- ES5 added one more way of creating an object
- `Object.create` takes `object1` as an argument and returns `object2` whose prototype is `object1`
- Allows to create objects with user-made prototypes without creating constructors

```javascript
var lannister = {name: 'Lannister'};
var swordsman = Object.create(lannister);
`Object.getPrototypeOf(swordsman) === lannister`;
true
swordsman.name
"Lannister"
```
Why bother?

- Why do we even bother with JavaScript?
- Because JavaScript allows us to work with the DOM
Document Object Model

- The Document Object Model (DOM) is a cross-platform and language-independent application programming interface that treats an HTML, XHTML, or XML document as a tree structure wherein each node is an object representing a part of the document. The objects can be manipulated programmatically and any visible changes occurring as a result may then be reflected in the display of the document. [1]

Document Object Model

Bioinformatics Resources

Hello!

We have a paragraph

and several `div`'s:
1.
2.
3.
Tree Structure

- Each element is a node in the DOM tree
- Each document element has an address
Traversing the DOM Tree

- `document` is the root element of the tree
- for each element, we can get it’s children with `childNodes`
Traversing the DOM Tree

- for each element, we can get it’s parent with `parentNode`
Manipulating DOM Tree

- We can retrieve, change, add, and remove elements
Retrieving DOM Elements

- `getElementById` returns an element with a specified id or `null`.
- This is too verbose for such a frequent operation.
- More compact way with jQuery – `$('#hello')`

```html
<document.getElementById('hello');
> <div id="hello">Hello!</div>
< $("#hello")
> <div id="hello">Hello!</div>
```
Retrieving DOM Elements

- `getElementsByTagName` returns an array of elements with a specified tag
  
  jQuery version – `$('tag')`

- `getElementsByName` returns an array of elements with a specified name (names are used on form elements)
  
  jQuery version – `$(name='name')`
Changing DOM Elements

- `innerText` property allows to change content of an element
- jQuery version – `$('selector').text('new text')`

```html
< document.getElementById('hello').innerText = 'Hi there!'
> "Hi there!"
< $('#hello').text('Hello!')
> [
>   <div id="hello">Hello!</div>
> ]
```
Creating DOM Elements

- Create an element with `createElement` and insert it into the DOM with `appendChild`.
- jQuery version – `$('selector').append($('<tag></tag>'));`
Removing DOM Elements

- `removeChild` removes a given element from an element it is called on
- jQuery version – `$('selector').remove`

```
< document.getElementById('third').parentNode.removeChild(document.getElementById('third'));
> <div id="third">3.</div>
< $('#third').remove();
> [<div id="third">3.</div>]
```