Jscript: an introduction

Lecture #2– A crash course in JavaScript

First things first

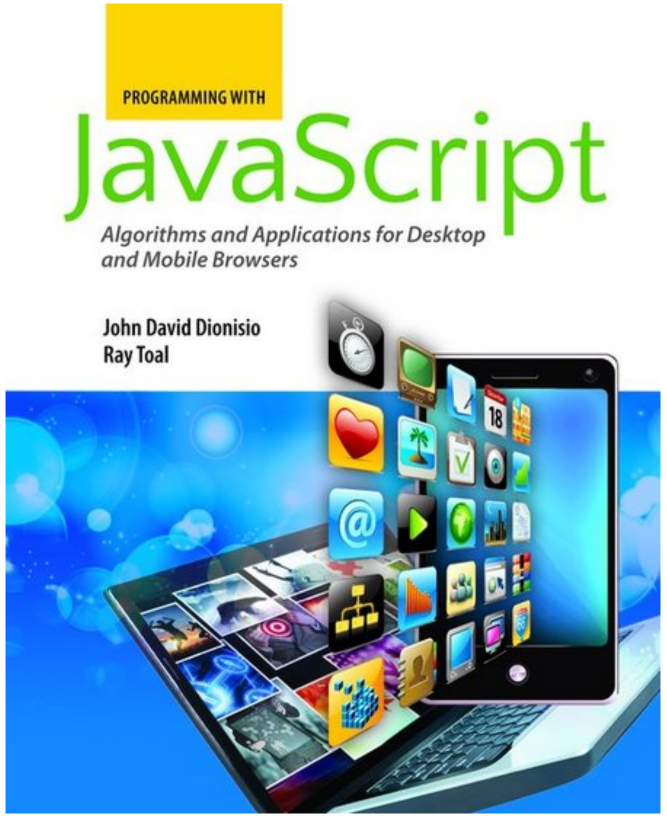
- JavaScript is not Java
- It is the most popular client-side scripting language
- It can be (and has been) misused
- It is rather unwieldy, which caused many toolkits and libraries to emerge
- It is object-based (but not purely OO)
- It can be used (extensively) for event-driven programming in web-based apps
- It is the 'J' in AJAX

Hands On!

- What do you need?
  - A browser! That's it.
- But we can do a bit better:
  - Firebug console
  - JavaScript Runner Page
  - JavaScript shell from squarefree.com
  - *Eloquent JavaScript* interactive book
  - Codecademy interactive JavaScript course
  - W3Schools JavaScript tutorial

Examples and exercises

- You can use the companion web site for the book “Programming with JavaScript: Algorithms and Applications for Desktop and Mobile Browsers”



- http://javascript.cs.lmu.edu/

Recipe for testing examples

1. Open example in a text editor
2. Open example in browser (+dev tools)
3. Run example (understand what it does)
4. Look at *how* it does it (using the dev tools)
5. Learn more about libraries, methods, built-in objects, etc.
6. Change the example to make it behave differently
7. Go back to 3

JavaScript Object Fundamentals

- In JavaScript, any value that is not a native data type (Boolean, number, string, *null* or *undefined*) is an Object.
- Objects have *properties*, and properties have *values*.
- An object literal is an expression defining a new object.
- Example:

```
var ride = {
   make : “Yamaha”,
   model :“V-Star Silverado 1100”,
   year : 2005,
   purchased = new Date(2005,3,12)
};
```

JavaScript Object Fundamentals

- After defining an object, you may access its properties with either a dot or square brackets.
- Example:

ride.make ➔ “Yamaha”

ride[“make”] ➔ “Yamaha”

JavaScript Object Fundamentals

- In JavaScript, the fundamental Object serves as the basis for all other objects. (similar to other languages)

- However, at its basic level, the JavaScript Object has little in common with the fundamental object defined by most other OO languages.

Creating a new Object

```
var shinyAndNew = new Object();
```

- But what can we do with this new object?
  - It seemingly contains nothing: no information, no complex semantics, nothing.
  - Our brand-new, shiny object doesn't get interesting until we start adding things to it—things known as *properties*.

Properties of objects

- Objects' properties / elements / data members can be created as needed.
- Example:

```
var ride = new Object();
ride.make = 'Yamaha';
ride.model = 'V-Star Silverado 1100';
ride.year = 2005;
ride.purchased = new Date(2005,3,12);
```

Properties of objects

- Flexibility comes with a price…
- Example:

```
ride.purchsaed = new Date(2005,3,12);
```

  – Will actually create a new property!

Objects and properties

- An instance of the JavaScript Object, or simply an *object*, is a collection of *properties*, each of which consists of a *name* and a *value*.
  - The *name* of a property is a string
  - The *value* can be any JavaScript object, be it a Number, String, Date, Array, basic Object, or any other JavaScript object type (including, as we shall see, functions).

Nested properties

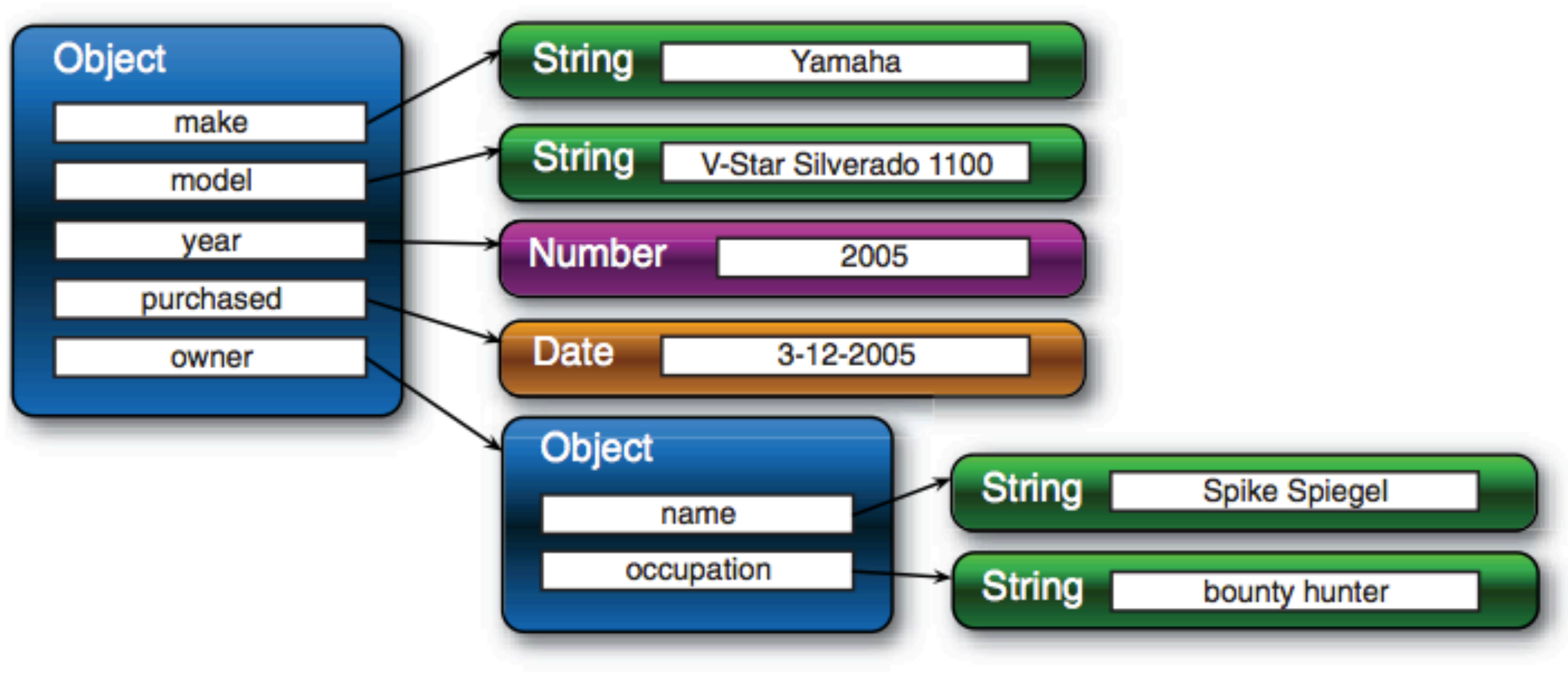
- Example:

```
var owner = new Object();
owner.name = 'Spike Spiegel';
owner.occupation = 'bounty hunter';
ride.owner = owner;
```

- To access the nested property, we write this:

```
var ownerName = ride.owner.name;
```

Object hierarchy



Property reference operator

- In many cases, the dot operator is inadequate, and we must use the more general notation for accessing properties.
- The format of the general property reference operator is:

  object[propertyNameExpression]

  – where *propertyNameExpression* is a JavaScript expression whose evaluation as a string forms the name of the property to be referenced.

Property reference operator

- Example - All three of the following references are equivalent:

```
ride.make
ride['make']
ride['m'+'a'+'k'+'e']
```

- So is this reference:

```
var p = 'make';
ride[p];
```

Object literals / JSON syntax

- Example:

```
var ride = {
  make: 'Yamaha',
  model: 'V-Star Silverado 1100',
  year: 2005,
  purchased: new Date(2005,3,12),
  owner: {
    name: 'Spike Spiegel',
    occupation: 'bounty hunter’
  }
};
```

Object literals / JSON syntax

- We can also express arrays in JSON by placing the comma-delimited list of elements within square brackets.
  - Example:

```
var someValues = [2,3,5,7,11,13,17,19,23,29,31,37];
```

Objects as window properties

- When the *var* keyword is used at the top level, outside the body of any containing function, it' s only a programmer-friendly notation for referencing a property of the predefined JavaScript *window* object.
- Any reference made in top-level scope is implicitly made on the window instance.

Objects as window properties

- All of the following statements, if made at the top level (that is, outside the scope of a function), are equivalent:

```
var foo = bar;
window.foo = bar;
foo = bar;
```

JavaScript Objects

- To summarize so far:
  - A JavaScript *object* is an unordered collection of *properties*.
  - Properties consist of a *name* and a *value*.
  - Objects can be declared using *object literals*.
  - Top-level variables are properties of *window*.

Functions as first-class citizens

- Functions in JavaScript are considered objects like any of the other object types that are defined in JavaScript, such as Strings, Numbers, or Dates.
- Like other objects, functions are defined by a JavaScript constructor — in this case Function — and they can be:
  - Assigned to variables
  - Assigned as a property of an object
  - Passed as a parameter
  - Returned as a function result
  - Created using literals
- Because functions are treated in the same way as other objects in the language, we say that functions are first-class objects.

Functions as objects

- And perhaps the trickiest part…
  - As with other instances of objects—be they Strings, Dates, or Numbers—functions are referenced only when they are assigned to variables, properties, or parameters.
  - Contrast the two snippets below:

```
function doSomethingWonderful() {
  alert('does something wonderful');
}
```

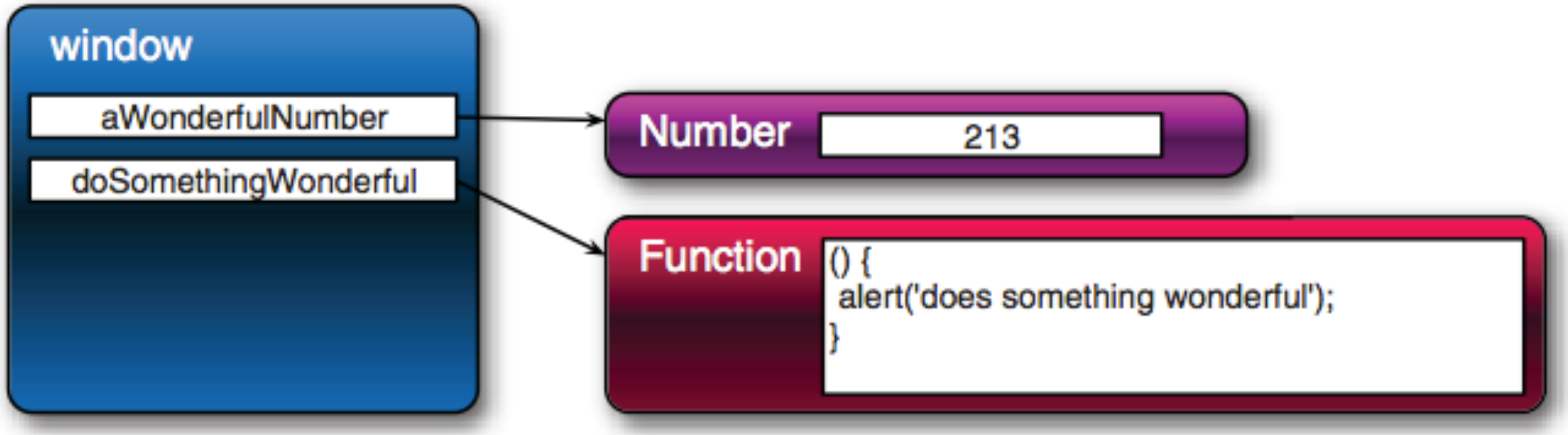
```
doSomethingWonderful = function() {
  alert('does something wonderful');
}
```

Functions as objects

- If we add the statement:

```
aWonderfulNumber = 213;
```

- A graphical representation would look like this:



```
() {
 alert('does something wonderful');
}
```

Functions as objects

- Therefore, the following three statements are equivalent:

```
function hello(){ alert('Hi there!'); }
hello = function(){ alert('Hi there!'); }
window.hello = function(){ alert('Hi there!'); }
```

- Important: Function instances are values that can be assigned to variables, properties, or parameters just like instances of other object types.
  - And like those other object types, nameless disembodied instances aren't of any use unless they're assigned to a variable, property, or parameter through which they can be referenced.

Callback functions

- Example:

```
function hello() { alert('Hi there!'); }

setTimeout(hello,5000);
```

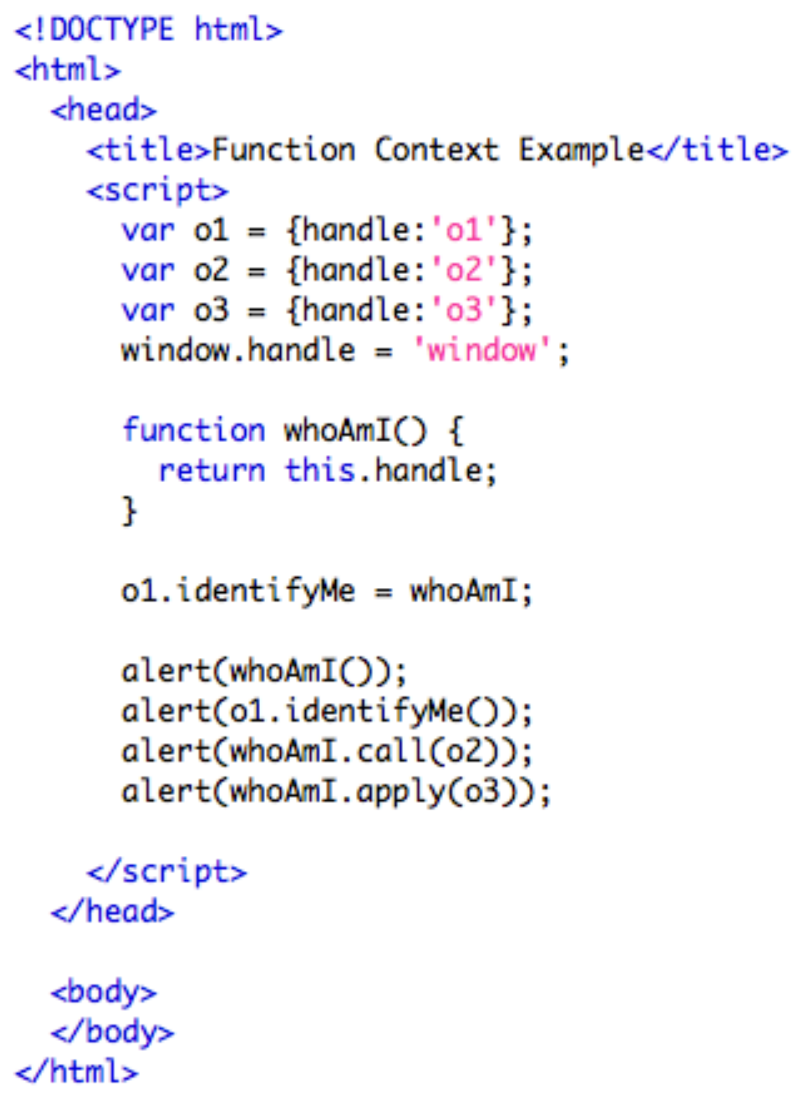
- Better version (used when there is no need for a function instance to be assigned to a top-level property:

```
setTimeout(function() { alert('Hi there!'); },5000);
```

Function context

- A more comprehensive example:
  - Can you tell which alert messages will be displayed each time?
  - What happens if we add a 5th alert:

```
alert(o1.identifyMe.call(o3));
```



```
<!DOCTYPE html>
<html>
  <head>
    <title>Function Context Example</title>
    <script>
      var o1 = {handle:'o1'};
      var o2 = {handle:'o2'};
      var o3 = {handle:'o3'};
      window.handle = 'window';

      function whoAmI() {
        return this.handle;
      }

      o1.identifyMe = whoAmI;

      alert(whoAmI());
      alert(o1.identifyMe());
      alert(whoAmI.call(o2));
      alert(whoAmI.apply(o3));

    </script>
  </head>

  <body>
  </body>
</html>
```

Function context

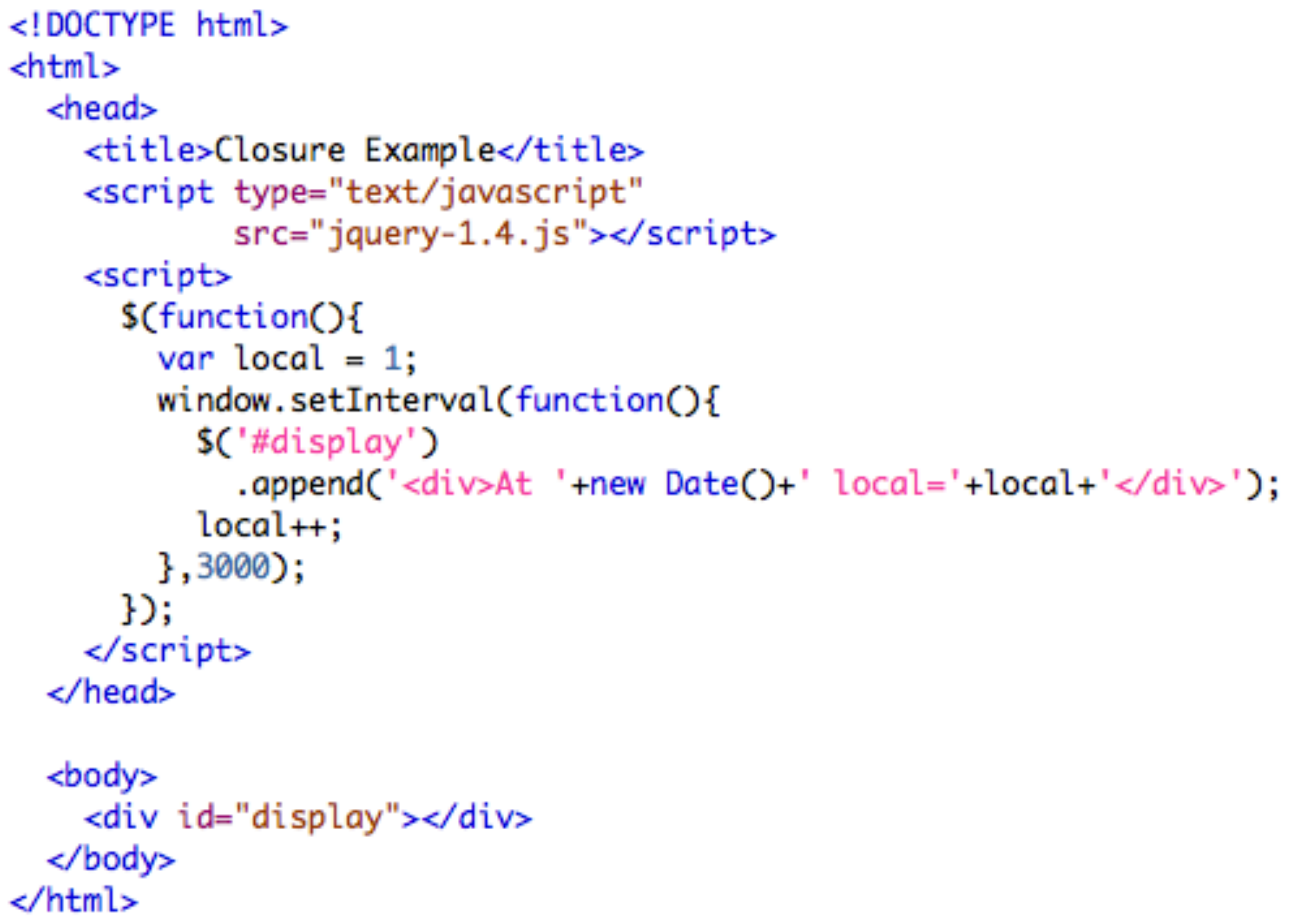
- This example page clearly demonstrates that the function context is determined on a per invocation basis and that a single function can be called with any object acting as its context.
- As a result, it' s probably never correct to say that a function is a method of an object.
- It' s much more correct to state:
  - *A function f acts as a method of object o when o serves as the function context of the invocation of f.*

Closures

- A *closure* is a Function instance coupled with the local variables from its environment that are necessary for its execution.
- When a function is declared, it has the ability to reference any variables that are in its scope at the point of declaration. (no surprises here)
- But, with closures, these variables are carried along with the function *even after* the point of declaration has gone out of scope, closing the declaration.
- The ability for callback functions to reference the local variables in effect when they were declared is an essential tool for writing effective JavaScript.

Closures

- Example



```
<!DOCTYPE html>
<html>
  <head>
    <title>Closure Example</title>
    <script type="text/javascript"
            src="jquery-1.4.js"></script>
    <script>
      $(function(){
        var local = 1;
        window.setInterval(function(){
          $('#display')
            .append('<div>At '+new Date()+' local='+local+'</div>');
          local++;
        },3000);
      });
    </script>
  </head>

  <body>
    <div id="display"></div>
  </body>
</html>
```

Closures

- It works! But how?
- Although it is true that the block in which *local* is declared goes out of scope when the ready handler exits, the closure created by the declaration of the function, which includes *local*, stays in scope for the lifetime of the function.

Closures

- Another example
  - Contrast this:

```
var withParentheses = function (s) {return "(" + s + ")";};
var withBrackets = function (s) {return "[" + s + "]";};
var withBraces = function (s) {return "{" + s + "}";};
```

  - With this:

```
var delimitWith = function (prefix, suffix) {
    return function (s) {return prefix + s + suffix;}
};

var withParentheses = delimitWith("(", ")");
var withBrackets = delimitWith("[", "]");
var withBraces = delimitWith("{", "}");
```

Events

- A programming paradigm shift
- Defining UI elements
- (Programmatically) accessing UI elements
  - The DOM
- Event Handling
- Event Objects and implementation details
- Case study: Tic-Tac-Toe