Advanced JavaScript and AJAX

Mathias Schwarz

(some slides by Anders Møller, Michael Schwartzbach)
(Some slides inspired by Douglas Crockford)

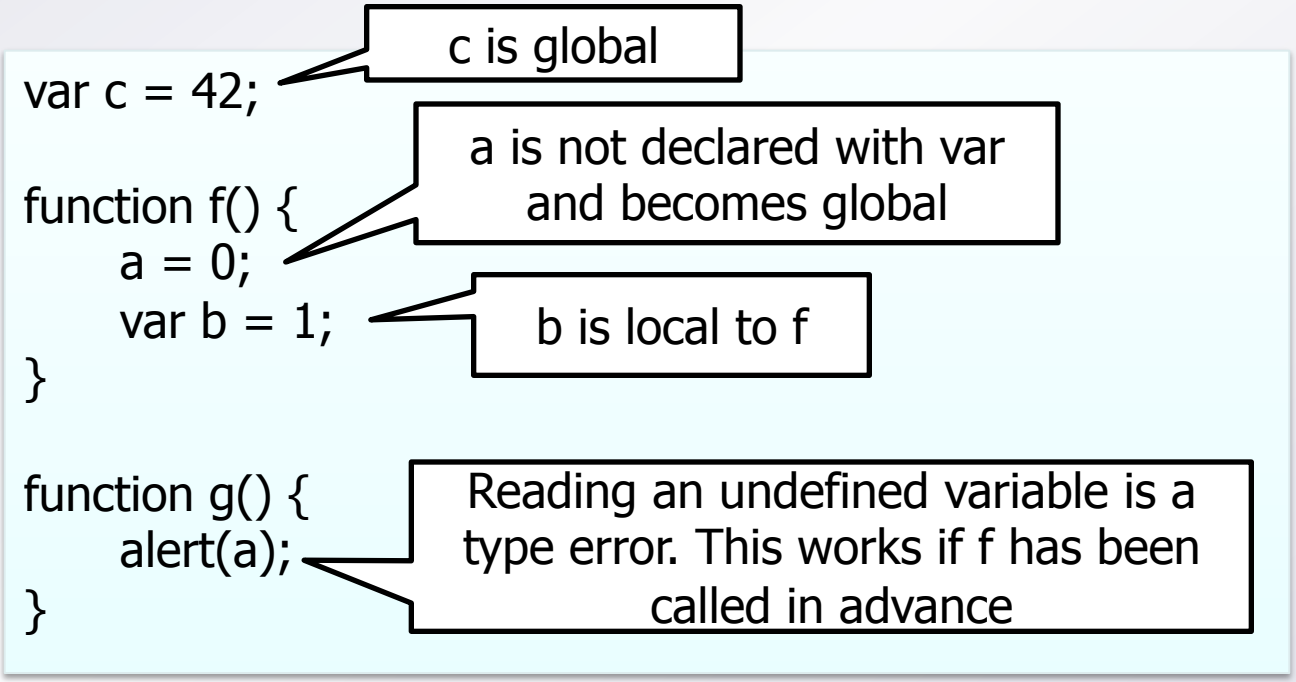
Agenda

- **Scope** and **closures** in JavaScript
- A closer look at **inheritance**
  - A recap of prototypes and class emulation
  - Emulation of prototype-based inheritance
  - Class-less inheritance
- Programming patterns in JavaScript

- **AJAX**
- **Guidelines** for better JavaScript code

Variable scope (1/2)

- Variables may be local to the function or global:



Variable scope (2/2)

- Variables declared with **var** are in scope of the whole function (!):

```
function f () {
        if (condition) {
                var b = 1;
        }
        alert (b);
}
```

We can refer to b here. If the value is set above, b is 1, otherwise it is undefined

- A variable can be declared twice so even if b is declared again there is only one b

The 'let' Statement

- The "let" statement (creates its own local scope block):

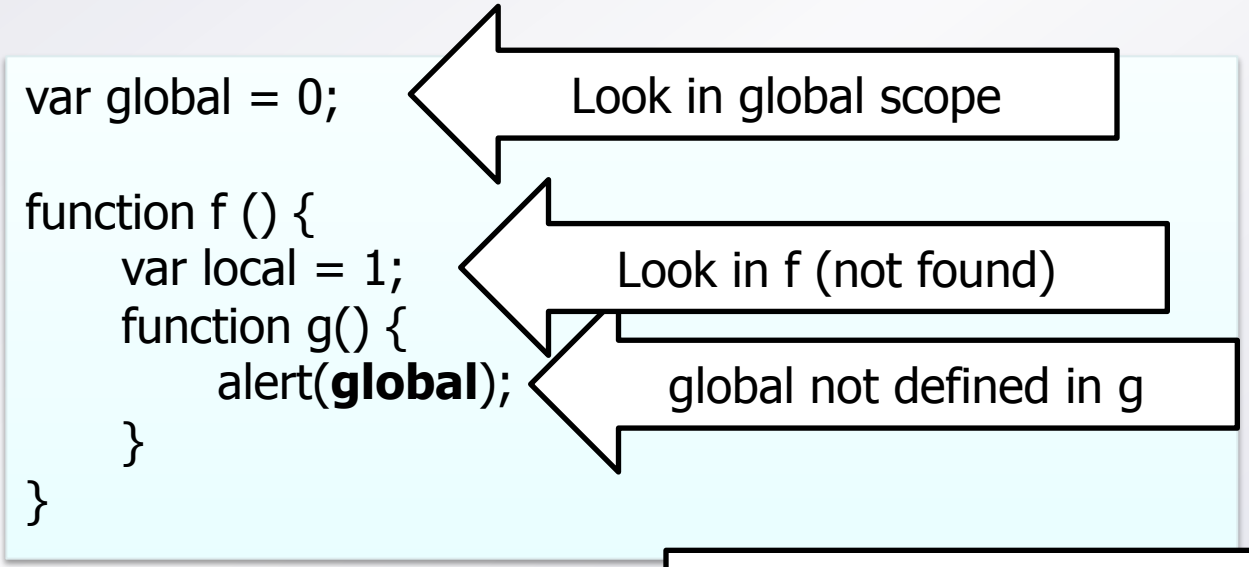
```
var x = 1;
var y = 2;

let (x = x+3, y = 4) {
  print((x + y) + "\n");
}

print((x + y) + "\n");
```

- ...will print out:

```
8
3
```

Lexical function scope

- In 99% of the cases we look up variables in the lexical scope:



The list of places to look things up is called the '**Scope Chain**'

The 'with' statement adds objects dynamically to this chain

Closures

- Functions "close over" the variables they refer to (i.e. keep their scope chain context), when returned:

```
function plusN(n) {
   return function(a) {
        return a + n;
   }
}
var plusTwo = plusN(2);
var plusFive = plusN(5);

var result1 = plusTwo(7);
var result2 = plusFive(7);
```

```
plusTwo is now:
function (a) {
    return a + n;
}
where the scope chain
contains a binding of n
to 2
```

```
result1 is 9
result2 is 12
```

A classical closure pitfall

- Since variables are scoped in functions, not blocks, the value of a variable is shared for all closures created in a function:

```
var addFunctions = new Array();

for (var i = 0; i < 10; i++) {
   addFunctions.push(function(a) {
      return i + a;    // i is shared
   });
}

addFunctions[1](42)    // yields 52, not 43(!)
```

A classical closure pitfall

- The solution is simple. Wrapping the closure in a function call gives a "fresh" (unshared) variable:

```
var addFunctions = new Array();

function addToFunctions(i) { // function remembers i
   addFunctions.push(function(a) {
      return i + a;
   });
}

for (var i = 0; i < 10; i++) {
   addToFunctions(i);
}

addFunctions[1](42)     // now yields 43(!), not 52
```

Functions in JavaScript

- Functions serve many purposes in JavaScript:
  - Running code
  - Methods on objects
  - Constructors for objects
  - **Pseudo Classes**
  - **Modules**

- We will now look more at the last two

A closer look at inheritance

- **Recall:** All objects have a prototype
- The prototype of an object can be set on the function used as constructor for the object
  - The "__proto__" pointer cannot be changed after construction (set to "prototype", at construction time)

```
function Student(name) {
  this.name = name;
}
Student.prototype.toString = function() {
   return this.name;
}

var stud = new Student("John");
var result = "Hello " + stud;
```

toString() is called when converting to a string

"Hello John"

instanceof

- The syntax is chosen to look like Java
  - Makes it possible to create 'pseudo classes':

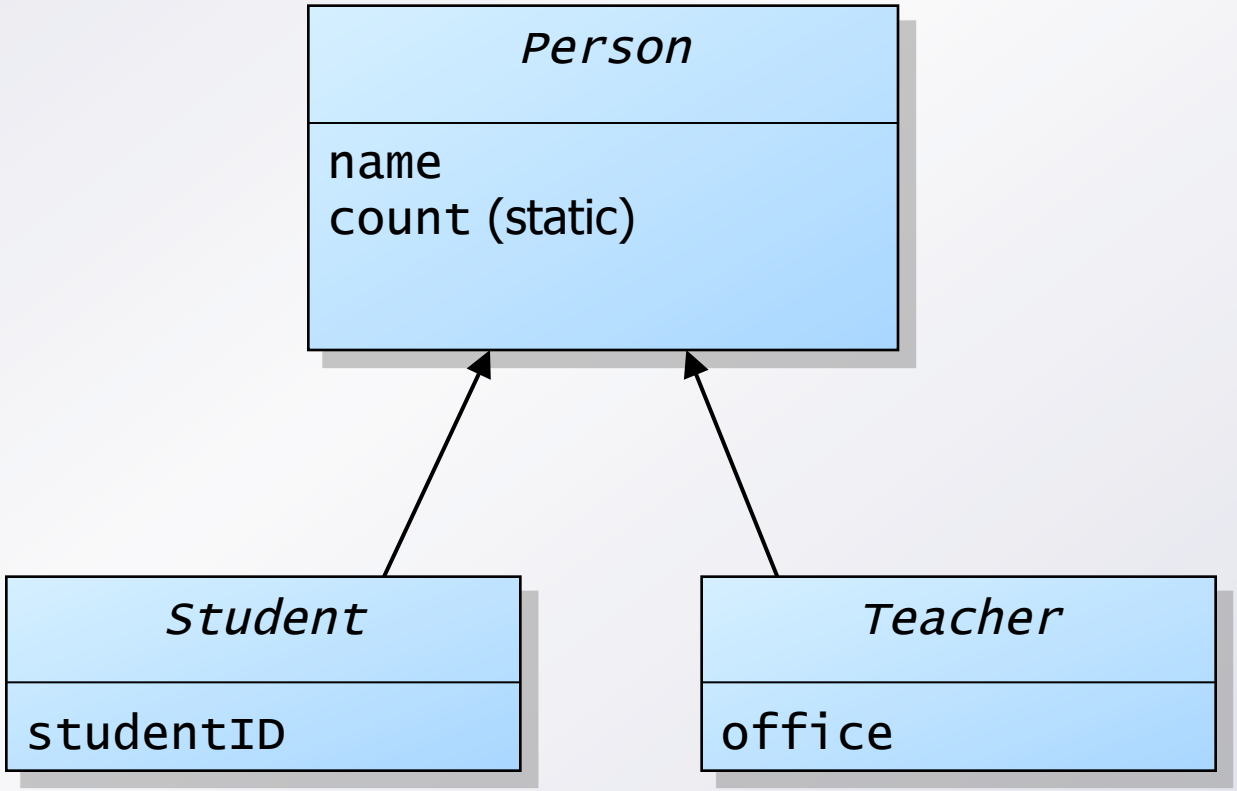
```
function Person(name) {
  this.name = name;
}
var stud = new Person("John")

if (stud instanceof Person) {
   …
}
```

instanceof checks if the objects is constructed by the given function

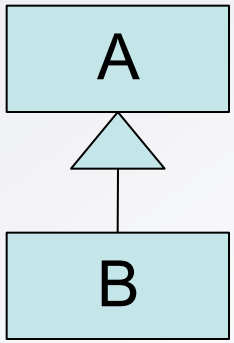
  - In fact the objects and prototypes are much more dynamic and expressive in JavaScript

Recap of class inheritance

- Combining pseudo classes with prototypes:
  - Pseudo class inheritance!



Pseudo class inheritance

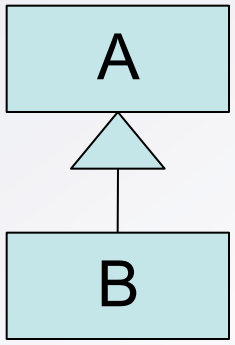


- Idea:
  - Throw away the prototype of a constructor function B
  - Replace it with an instance of an object created by another constructor A
  - Thus when looking up properties on B:
    - First look a properties on B
    - If not found, look at B's prototype (that is the A object)
  - This looks somewhat like inheritance in Java (A being super class of B)

Inheritance (B extends A)

```
function A(a) {
  this.a = a;
}

function B(a, b) {
  this.foo = A; // super
  this.foo(a);  // super
  delete this.foo; //
  this.b = b;
}

B.prototype = new A();

var x = new B(42,87);

x.b
```

87

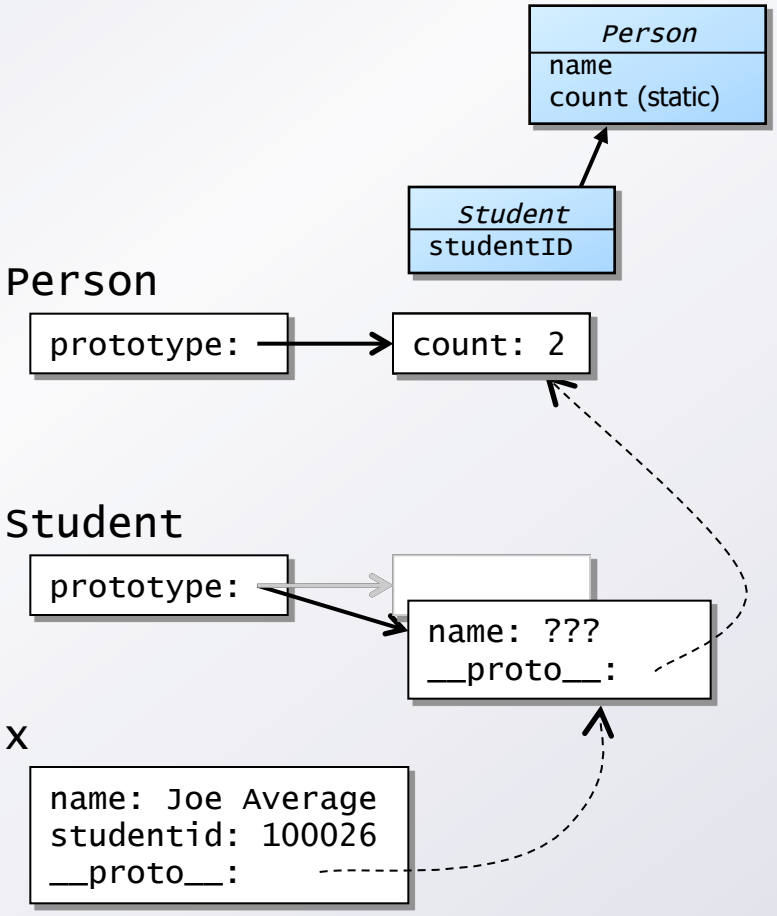



```
function A(a) {
  this.a = a;
}

function B(a, b) {
  A.call(this,a); //super
  this.b = b;
}

B.prototype = new A();

var x = new B(42,87);

x.b
```

87

Adding the static count field

```
function Person(n) {
  this.name = n || "???"
  Person.prototype.count++
}
Person.prototype.count = 0

function Student(n,s) {
  Person.call(this,n);
  this.studentid = s
}
Student.prototype = new Person
```




```
var x = new Student("Joe Average", "100026")
print(x.count) // returns 2
```

Prototype-based inheritance

- Classical prototype-based languages:
  - Create new objects by copying an existing one (hence the name "prototype")
    - We can copy an object by iterating over its properties (e.g., using "for … in" statement)
  - A completely class-less model
  - Such languages have an *object* (xerox) func:
    - Makes a new object with an existing object as its prototype
  - The JavaScript model can emulate this as well!

Prototype-based inheritance

- Create a local function that:
  - Has the existing object as prototype
  - Is used as constructor for the new object:

```
function object(existing) {
   function Temp() {};
   Temp.prototype = existing;
   return new Temp();
}
```

if a constructor returns an object, the returned object is the result!

  - Since this is a local function, a new instance of 'Temp' is created each time:

```
var newObject = object(existing);
```

'new' can be left out

Prototype-based inheritance (1/2)

```
function object(existing) {
    function Temp() {};
    Temp.prototype = existing;
    return new Temp();
}

var x = { a: 87, b: "str" };

var y = object(x);
```

```
js> x.a = 42;
js> y.a
42              // NB: sharing !
```

```
js> y.a = 42; // shadowing!
js> y.a
42

js> delete y.a;
js> y.a
87
```

Prototype-based inheritance (2/2)

```
function object(existing) {
    function Temp() {};
    Temp.prototype = existing;
    return XEROX(new Temp());  // EXERCISE: xerox object !
}

var x = { a: 87, b: "str" };

var y = object(x);
```

```
js> y.a
87
js> x.a = 42;
js> y.a
87                              // still 87, not 42!
```

Prototype-based inheritance (2/2)

```
function object(existing) {
    function Temp() {};          // empty template
    Temp.prototype = existing; // set 'prototype'
    var New = new Temp();        // create new empty object and ...
    for (var x in existing) {  // ... xerox !
        New[x] = existing[x];
    }
    return New;                  // return new xerox'ed object
}

var x = { a: 87, b: "str" };

var y = object(x);
```

```
js> y.a
87
js> x.a = 42;
js> y.a
87                               // still 87, not 42!
```

Private members in objects

- Private members are possible in JavaScript:
  - Properties of objects are all externally visible
  - Free variables in closures are not**(!)**
- We want to extend the Person objects with a private ID property
  - And return it from a method on Person

Private members in objects

- **Idea:** Use a closure to hold the variable (and use "getters" and "setters"):

```
function Person(name) {
  this.name = name;
  var ID;                    // free variable, not a property!
  this.getID = function() { return ID }          // getter!
  this.setID = function(newid) { ID = newid }    // setter!
}

var me = new Person("Mathias");
```

```
js> me.setID(42)
js> me.getID()
42
```

Modules

- The global environment gets crowded in large programs
  - Java has packages to structure programs
  - C# has namespaces
  - …
- In JavaScript, we can use objects (and functions) as modules

Modules

- **Simple idea:** Use an object literal!

```
var iwjx = {                                    // module “iwjx”
  addOne: function(n) { return n + 1 },
  addTwo: function(n) { return n + 2 }
}

var other = {                                   // module “other”
  addOne: function(n) { return n + ”1” }
}
```

```
js> iwjx.addOne(42)
43
js> other.addOne(42)
421
```

- In this model all members are **public**

Functions as modules

- We can use a function to create a module with a **private** module member!

```
var iwjx = new function() {
  var addN = function(n) {                    // private function
    return function(a) { return a + n };
  }
  return { addOne: addN(1), addTwo: addN(2) };
};
```

```
js> iwjx.addOne(42)
43
js> iwjx.addN(42)
undefined
```

- This is a must for large programs

Conclusions on JavaScript

- Inheritance model based on:
  - Prototype link that is used for property lookup
- The JavaScript inheritance model is powerful:
  - Can somewhat emulate class-based inheritance
  - … and prototype-based inheritance
- Functions provide types, encapsulation, modules and pseudo classes

Agenda

- Scope and closures in JavaScript
- A closer look at inheritance
  - A recap of prototypes and class emulation
  - Emulation of prototype-based inheritance
  - Class-less inheritance
- Programming patterns in JavaScript

- **AJAX**
- **Guidelines for better JavaScript code**

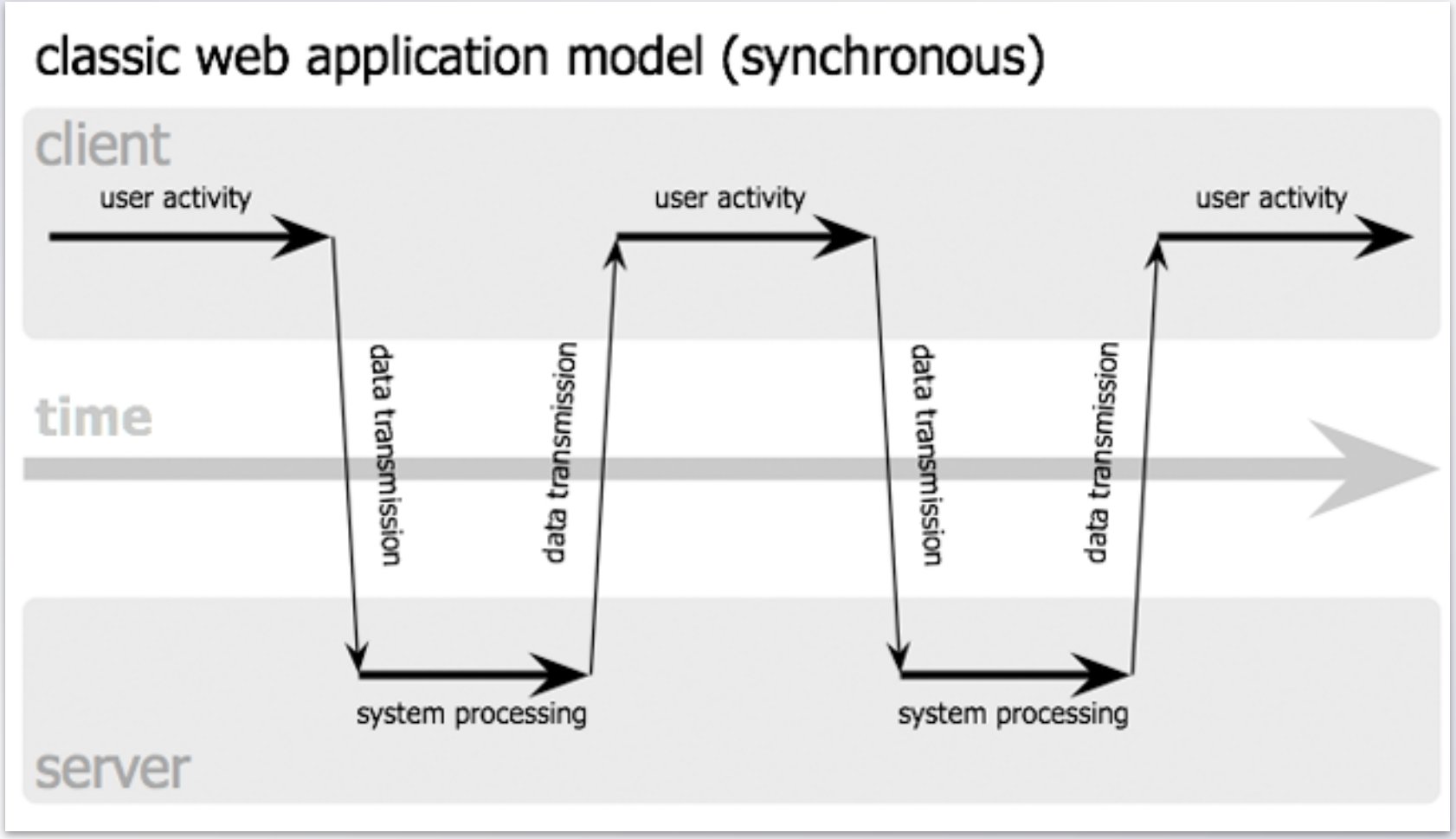
AJAX

AJAX = **A**synchronous **J**avaScript **a**nd **X**ML

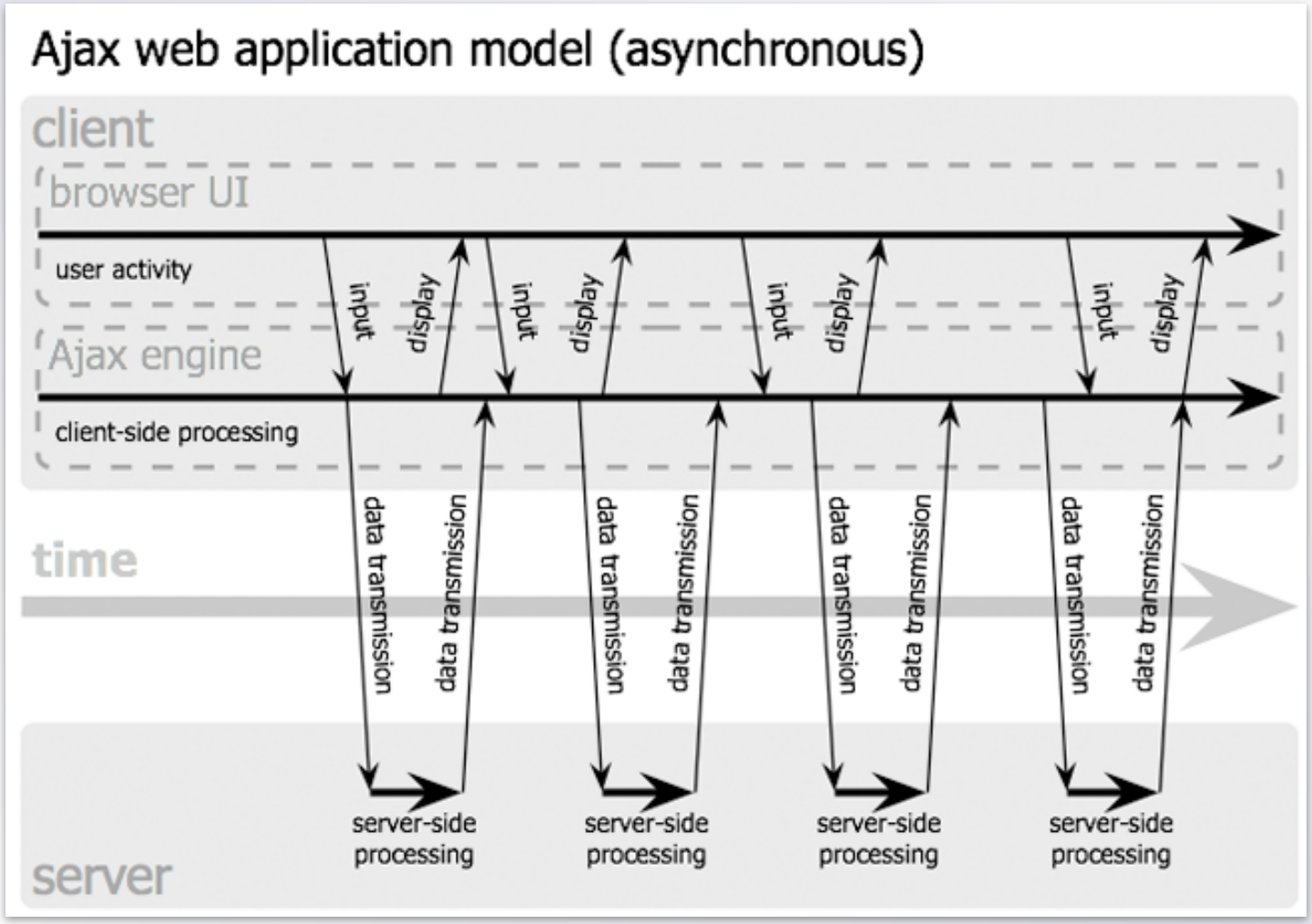
- Extends the traditional request–response cycle from HTTP
- Allows JavaScript code on the client to issue HTTP requests back to the server

- The core of AJAX (the `XMLHttpRequest` / `XMLHTTP` object) is being standardized by W3C:
  `http://www.w3.org/TR/XMLHttpRequest/`

AJAX


classic web application model (synchronous)
client
user activity
user activity
user activity
data transmission
data transmission
data transmission
data transmission
time
system processing
system processing
server

AJAX


Ajax web application model (asynchronous)
client
browser UI
user activity
input
display
Ajax engine
client-side processing
data transmission
time
server-side processing
server

AJAX example

A primitive AJAX library:

ajax.js:

```
var http;
if (navigator.appName == "Microsoft Internet Explorer")
  http = new ActiveXObject("Microsoft.XMLHTTP");
else
  http = new XMLHttpRequest();

function sendRequest(action, responseHandler) {
  http.open("GET", action);
  http.onreadystatechange = responseHandler;
  http.send(null);
}
```

(HTTP *persistent connections* are not generally exploited yet)

Simple AJAX test

- Simple test:

readyState (status of the XMLHttpRequest):
0: request not initialized
1: server connection established
2: request received
3: processing request
4: request finished and response is ready

```
var http = new XMLHttpRequest();  // different on MSIE

function sendRequest(action, responseHandler) {
  http.open("GET", action);
  http.onreadystatechange = responseHandler;
  http.send(null);
}

sendRequest("http://www.eb.dk", function () {
  alert(http.readyState)
});
```

2
3
3
4

innerHTML

- Text can be parsed as HTML and inserted in the DOM
  - Internet Explorer extension
  - Is being standardized and works in all browsers

```
var click = document.createElement("b");
var text = document.createTextNode("Click me");
click.appendChild(text);
element.appendChild(click);
```

  - …vs…:

```
element.innerHTML = "<b>Click me</b>”
```

AJAX example

ajax-demo.html (1/3):

```
<html>
<head>
<title>AJAX demo</title>
<script type="text/javascript" src="ajax.js"></script>
<script type="text/javascript">
var timeout;

function buttonClicked() {
  if (http.readyState == 0 || http.readyState == 4) {
    // the request object is free
    var v = document.getElementById("x").value;
    sendRequest("http://www.brics.dk/ixwt/echo?X="+
                encodeURIComponent(v), responseReceived);
  } else { // let's try again in .5 sec
    window.clearTimeout(timeout);
    timeout = window.setTimeout(buttonClicked, 500);
  }
}
```

AJAX example

ajax-demo.html (2/3):

```
function responseReceived() {
  if (http.readyState == 4) // operation completed?
    try {
      if (http.status == 200) { // OK?
        var d = document.createElement("div");
        d.innerHTML = http.responseText; // parse HTML
        var t = d.getElementsByTagName("table")[0]; // extract table
        var r = document.getElementById("result");
        r.replaceChild(t, r.firstChild);
      } else
        alert("Error " + http.status);
    } catch (e) { // may occur in case of network error
      alert(e);
    }
}
</script>
```

AJAX example

ajax-demo.html (3/3):

```
<body>

Enter some text:
<input name="text" id="x" onkeyup="buttonClicked()">
<p>
<div id="result">
no data yet...
</div>

</body>
</html>
```

http://www.brics.dk/ixwt/echo?x=y

http://www.brics.dk/~amoeller/AWT/ajax-demo.html

Client-side security

- No access to client file system etc.
- **Same-origin policy:**
  - a script can read only the properties of windows and documents that have the same origin as the document containing the script
  - `XMLHttpRequest` can only contact an origin server (to prevent impersonation)
- **Cross-site scripting (XSS):**
  - server vulnerability that permits attackers to inject scripts in other web sites (typically caused by lack of input validation)

» see http://en.wikipedia.org/wiki/Samy_(XSS)

JSON

– ***Java*S*cript* O*bject* N*otation***

- A light-weight alternative to XML for data interchange (in particular, of JavaScript values) – typically used with AJAX
- A simple textual representation for **acyclic(!)** JavaScript data structures
- See http://www.json.org/

JSON Example

- Now you are JavaScript experts so this requires little presentation:

```
var str_person =
  '({"name": "John Doe", "age": 13, "has_kids": false})'
var json_person = eval(str_person)
print(json_person.name)
```

- *eval* can however run any code in str_person, so a specialized JSON eval (called JSON.parse) is coming

JavaScript best practices

- Selected recommendations (Best Practices):
  - Avoid Globals
    (to avoid cluttering up global environment)
  - Modularize
  - Stick to a strict coding style
    (do no rely on browser auto-corrections!)
  - Detect features, not browsers
  - Validate all data
    (to avoid XSS)

Tool support for improved code

- There is little more to say about JavaScript
- Even great programmers make errors
  - JavaScript has many subtle pitfalls
- It takes years to master this language
  - Tool support is needed...

JSLint

- Lint was a bug finding tool for C
  - Simple detection of common problems
- JSLint is for JavaScript what Lint was for C
- Detects a wide variety of bugs, including:
  - Misuse of globals (often spelling errors)
  - Error-prone line breaking
  - Scope problems (like the one we saw)
  - Unreachable code
  - …
  - http://www.jslint.com/lint.html

TAJS

- **T**ype **A**nalyzer for **J**ava**S**cript
- Inference of type information
- Can detect type errors such as:
  - Invoking something that is not a function
  - Reading absent variables (and properties)
  - Error prone coercions
  - …

Online resources

- JavaScript 1.5 Guide: http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide
- ECMAScript Specification: http://www.ecma-international.org/publications/standards/Ecma-262.htm
- Douglas Crockford's page: http://javascript.crockford.com/
- Best practices: http://dev.opera.com/articles/view/javascript-best-practices/