Dynamic Languages World Europe 2008

Andreas Ecker | 1&1 Internet AG

Advanced Object-Oriented JavaScript

These keywords belong too ...

```
abstract  extends     package    throws
boolean   final       private    transient
byte      float       protected  volatile
char      implements  public
class     import      short
const     int         static
double    interface   super
enum      long        synchronized
export    native
```

... JavaScript !

- **ECMAScript 3 / JavaScript 1.5**
- Standard spezified in 1999
- Supported by all common browsers
- Many keywords *reserved*, but not used

- **ECMAScript 4 / JavaScript 2.0**
- Many new language features
- Not expected any time soon

Data Types

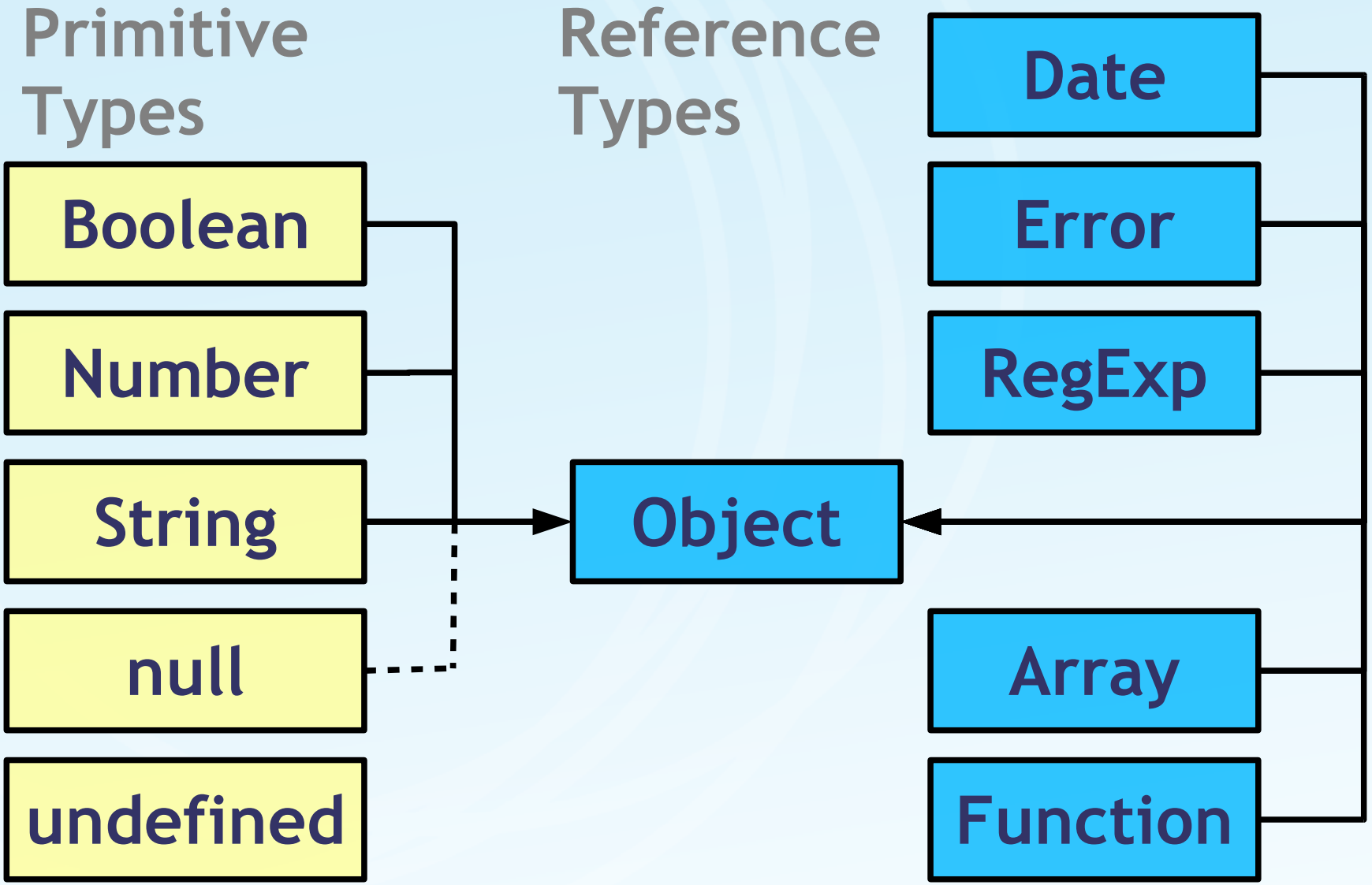
- Primitive Types
- Reference Types

Primitive Types

- Boolean `true, false`
- Number `1, 3.141, -1.602e-19`
- String `"Joe"`
- null `null`
- undefined `undefined`

Reference Types

- Date `new Date(1211623944453);`
- Error `new Error("Oops!");`
- RegExp `/^web.*$/i;`
- Array `[ "apple", "banana" ]`
- Function `function(x) {return x*x}`


Primitive Types
Reference Types
Boolean
Number
String
null
undefined
Object
Date
Error
RegExp
Array
Function

typeof

- Boolean `"boolean"`
- Number `"number"`
- String `"string"`
- null `"object"` Caution!
- undefined `"undefined"`
- Array `"object"` Caution!
- Function `"function"`
- Object `"object"`

Object

unordered collection of properties with arbitrary values

- object literal
```
var obj = { name: "Joe", age: 26 };
```
- setting a property
```
obj.lastName = "Smith";
```
- retrieving properties
```
alert(obj.name + " " + obj.lastName);
```

Object as Hash

Data structure that associates arbitrary values with arbitrary strings

- property name as an *identifier*

```
obj.lastName = "Smith";
```

- property name as a *string*

```
obj["lastName"] = "Smith";
```

-
```
for( prop in obj ) {
    alert( prop + ": " + obj[prop] );
}
```

Class

Concept of a *class* does not exist...

... but use a function as a *constructor*:

- `function Dog() {};` class “Dog”
- `var lassie = new Dog;` instance “lassie”
- `alert(lassie instanceof Dog); // true`

Static Members

Because functions are „first-class objects“ we can attach properties:

- Class Variables

```
Dog.SPECIES = "Canis lupus";
```

- Class Methods

```
Dog.getCount = function() {
   return Dog.COUNT;
};
```

Instance Members

- Instance Variables

```
function Dog(name) {
   this.name = name;
};

var lassie = new Dog("Lassie");
alert( lassie.name );
```

Instance Members

- Instance Methods

```
function Dog(name) {
   this.name = name;
   this.bark =
     function() { alert("Woof!") };
};

var lassie = new Dog("Lassie");
lassie.bark();
```

Scope

- Global Scope
  - Variables outside of any functions
  - Variables inside functions without `var`

```
var global1 = 1;
global2 = 2;

function foo() {
  global3 = 3;
};
```

Scope

- Function Scope
  - Variables inside functions declared with **`var`**
  - Function arguments

```
function foo(local1) {
  var local2 = 2;
};
```

Scope

- ~~Block Scope~~

... but can be faked:

```
// before block
(function() {

  // inside block

})();
// after block
```

Private Members

```
function Dog(name) {
   var _name = name;  // private variable
   // privileged method
   this.getName = function() {
      return _name;
   };
};

var lassie = new Dog("Lassie");
alert( lassie.getName() );
```

Private Members

```
function Dog(name) {
   var _name = name;
   // private method
   var _fixName = function() {
      return _name.toUpperCase();
   };
   this.getName = function(){
      return _fixName();
   };
};
```

Closures

- Nested functions
- Inner function has still access to local variables even after the outer function has finished

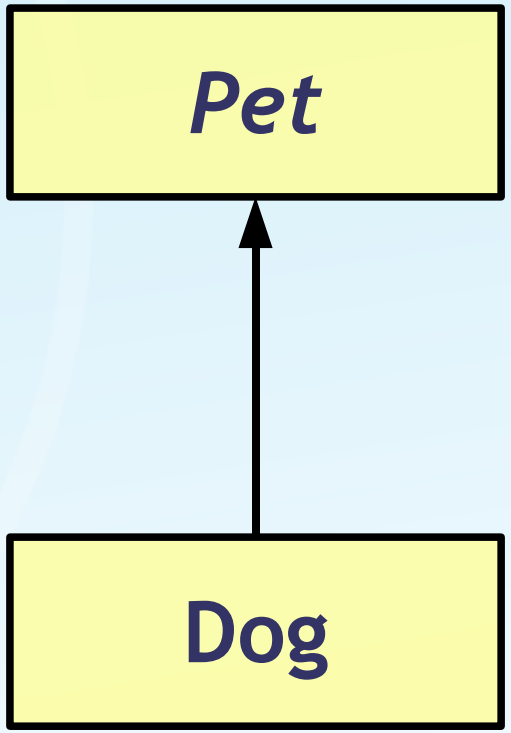
Closures

```
function outer()
{
  var count = 1;
  function inner() { alert(count++) }
  return inner;
}

var myClosure = outer();
myClosure();   // ==> 1
myClosure();   // ==> 2
```

Inheritance (native)

- `function Pet() {};`
- `function Dog() {};`
- `Dog.prototype = new Pet;`



Calling the superclass' constructor

```
function Pet(name) {
   this.name = name;
};

function Dog(name) {
   // super(name)
   Pet.call( this, name );
   this.bark = function() {};
};

Dog.prototype = new Pet;
```

Calling a Method with Arbitrary Scope

Inside the method `this` now refers to the scope you supplied:

- ```
  function foo() {
      this.bar();
  };
  ```
- `foo.call( scope, arg1, arg2, ... );`
- `foo.apply( scope, [arg1, arg2, ...] );`
  ```

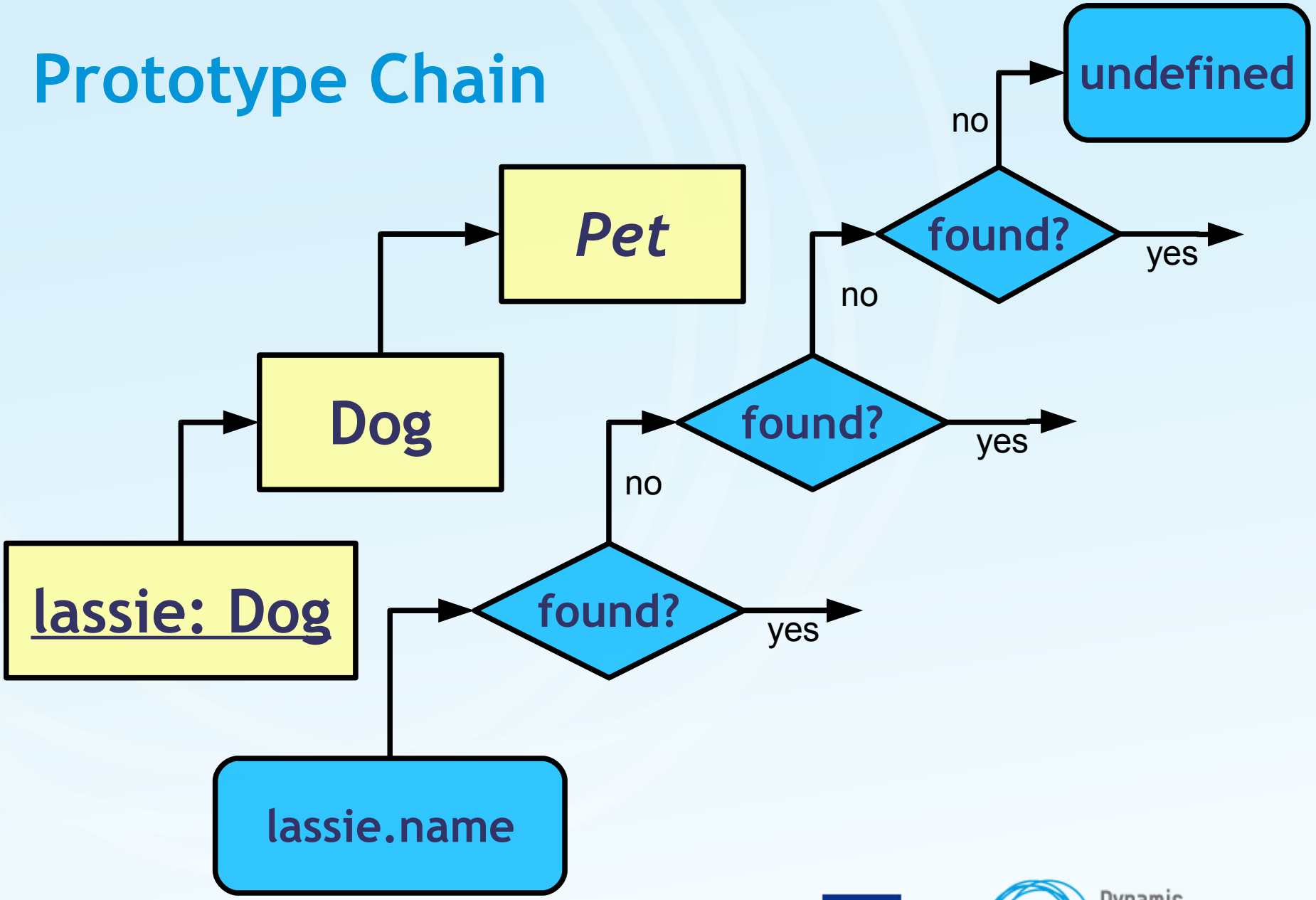
Instance Members (improvement)

- ```
  // old: attach to "this"
  function Dog(name) {
      this.bark =
          function(){ alert("Woof!") };
  };
  ```

- ```
  // new: attach to "prototype"
  function Dog(name) {};

  Dog.prototype.bark =
          function(){ alert("Woof!") };
  };
  ```

Prototype Chain



Instance vs. Prototype

- Property values on *instance*: local, instance-specific values
- Property values on *prototype*: read-only default values

Attaching to the prototype saves memory, especially for *large* numbers of instances

Class Augmentation

- Affects all new instances
- Affects all existing instances
- Allows modification of existing classes

```
String.prototype.trim =
   function() {
      return this.replace(/^\s+/, "");
   };

alert("   Lassie".trim() );
```

Overriding Methods

```
function Dog() {};
Dog.prototype.bark =
        function() { alert("Woof!") };

function Bulldog() {};
Bulldog.prototype = new Dog;

Bulldog.prototype.bark = function() {
  // super.bark();
  Dog.prototype.bark.call(this);
  alert("Grrrh!!") };
```

Abstract Class

```
function Pet() {
  if(this._id == Pet._id) {
    throw new Error("No Pets, please!");
  }
}
Pet._id = "Pet";
Pet.prototype._id = "Pet";

var fiffy = new Pet; // Error (intended)
```

Inheritance (improved)

But now our code to setup inheritance will fail:

```
Dog.prototype = new Pet;    // Error :-(
```

Solution: Do not create an instance of the *actual* superclass just to *setup* inheritance, use a dummy:

```
function Dummy() {};
Dummy.prototype = Pet.prototype;

Dog.prototype = new Dummy;
```

Namespaces

```
if (typeof very == "undefined") {
   very = {};
}
if (typeof very.cute == "undefined") {
   very.cute = {};
}

very.cute.Dog = function() {};
var fiffy  = new very.cute.Dog;
```

Singleton (“Essence”)

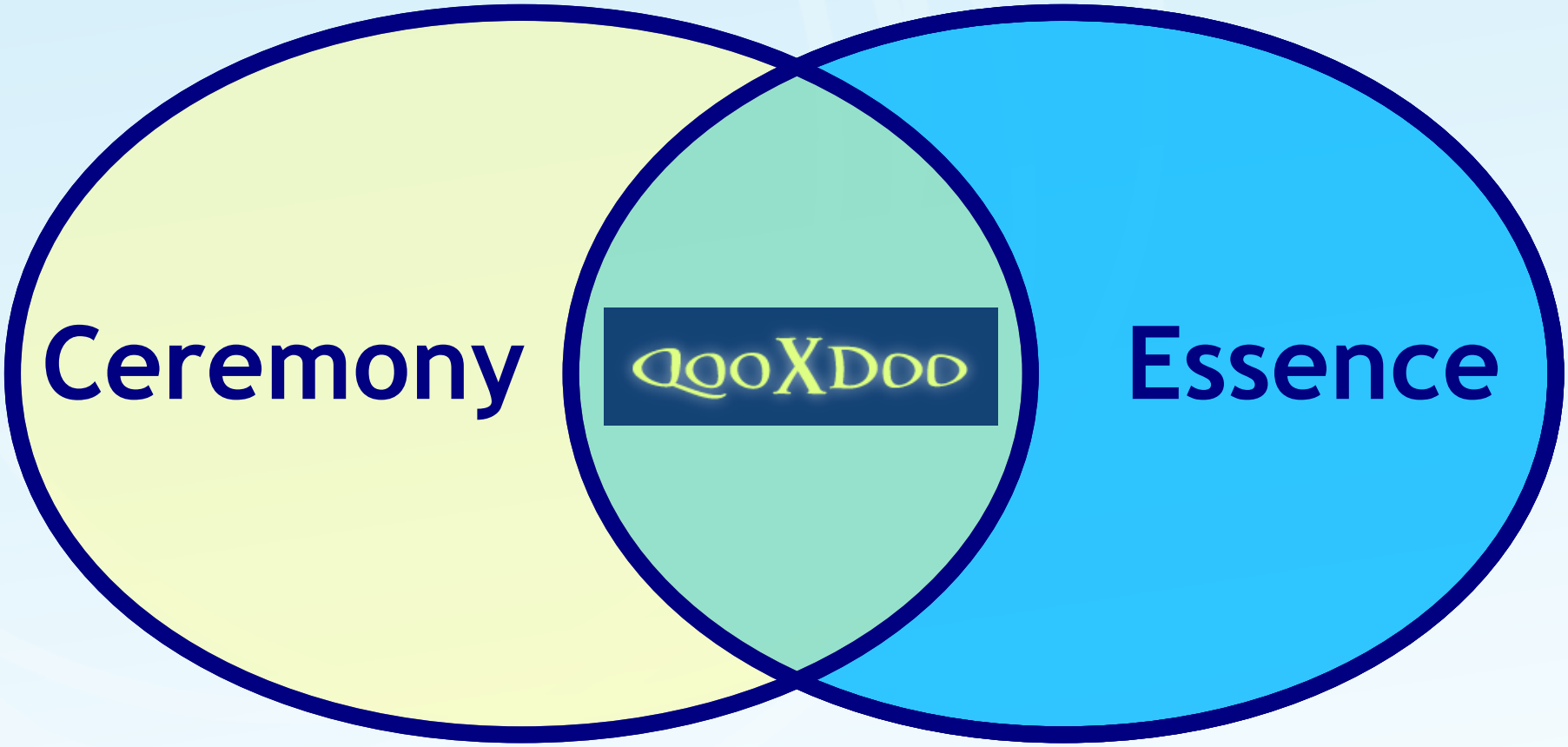
```
// The Last of the Mohicans
var chingachgook = {
   fight : function() {
      alert("Woah!");
   }
};

chingachgook.fight();
```

Singleton (“Ceremony”)

```
function Mohican() {
 this.fight = function(){alert("Woah!")}
};
Mohican.getInstance = function() {
   if (!this._instance) {
      this._instance = new this; }
   return this._instance;
};

Mohican.getInstance().fight();
```



OO Wishlist

- Closed form of class declaration
- Namespaces
- Static members
- Instance members
- Inheritance
- Superclass call (constructor, overriden methods)

http://qooxdoo.org

Open-Source JavaScript Framework

1&1

Dynamic Languages World Europe 2008

Class Declaration

```
qx.Class.define( "very.cute.Dog", {
  extend: Dog,
  construct: function(name) {
    this.base( arguments, name); },
  statics: {
    SPECIES: "Canis lupus familiaris" },
  members: {
    bark: function() { alert("wooof");}}
});
```

OO Wishlist (continued)

- Static class
- Abstract class
- Singleton

Static Class

```
qx.Class.define( "DogUtil", {
   type: "static",
   statics: {
      SPECIES: "Canis lupus",
      getCount: function() {}
   }
});
```

Abstract Class

```
qx.Class.define( "Pet", {
   type: "abstract",
   construct: function(name) {
      this.name = name;
   },
   members: {
      getName: function() {
         return this.name; }
   }
});
```

Singleton

```
qx.Class.define( "Mohican", {
   type: "singleton",
   members: {
      fight: function() {
         alert("Woah!");
      }
   }
});

var chingachgook = Mohican.getInstance()
chingachgook.fight();
```

OO Wishlist (continued)

- Destructors

Destructor

```
qx.Class.define( "very.cute.Dog", {
   construct: function() {
      this._toys = new some.complex.Toy;
   },
   destruct: {
      this._disposeObjects("_toys");
   }
});
```

OO Wishlist (continued)

- Getters
- Setters

Dynamic Properties

```
qx.Class.define( "very.cute.Dog", {
   properties: {
      name: { check: "String" }
   }
});

var fiffy = very.cute.Dog;
fiffy.setName("Fiffy");
alert( fiffy.getName() );
```

OO Wishlist (continued)

- **Interfaces**

  Definition of (empty) methods, that a class must implement individually

- **Mixins**

  Definition of (non-empty) methods, etc. that are added to an existing class

Interfaces

```
qx.Interface.define( "IVenomous", {
   members: {
      actionToKill: function() {} }
});
qx.Class.define( "Snake",
   implement: IVenomous,
   members: {
      actionToKill: function() {
         /* bite code here */ }
});
```

Mixins

```
qx.Mixin.define( "MMigrant", {
   members: {
      leave: function() { /* code! */ },
      return: function() { /* code! */ },
   }
});

qx.Class.define( "Stork",
   include: [MMigrant, MClattering]
});
```

Aspect-oriented Proprammming (AOP)

Attach advices before or after each function call

```
qx.core.Aspect.addAdvice(
   "before", // or "after"
   "*",      // or "member", "static",
             // "constructor",
             // "destructor", "property"
   "your.namespace.*",  // regexp
   myAdvice
);
```

Aspect-oriented Proprammming (AOP)

Signature of advice function:

```
myAdvice(
   fullName, // full function name
   target,   // function to call
   type,     // "before" or "after"
   args,     // arguments to target
   retValue  // only for "after" advice
);
```

Please try that at home ☺

- **`http://qooxdoo.org`**
- qooxdoo: OO Documentation
- qooxdoo: API Viewer
- Crockford: JavaScript Survey
- Crockford: Private Members
- Flanagan: JavaScript (O'Reilly)
- **`andreas.ecker@1und1.de`**