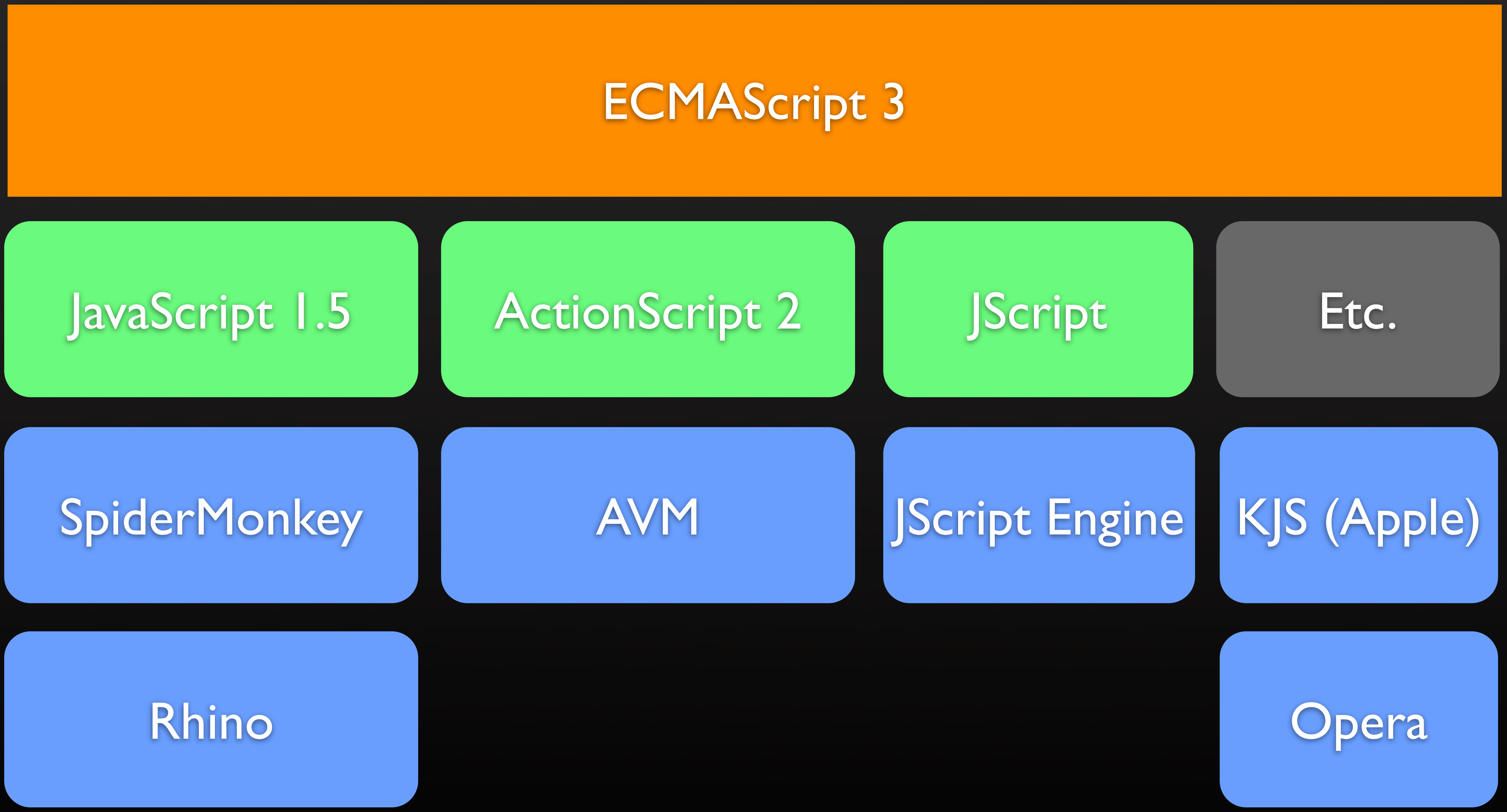
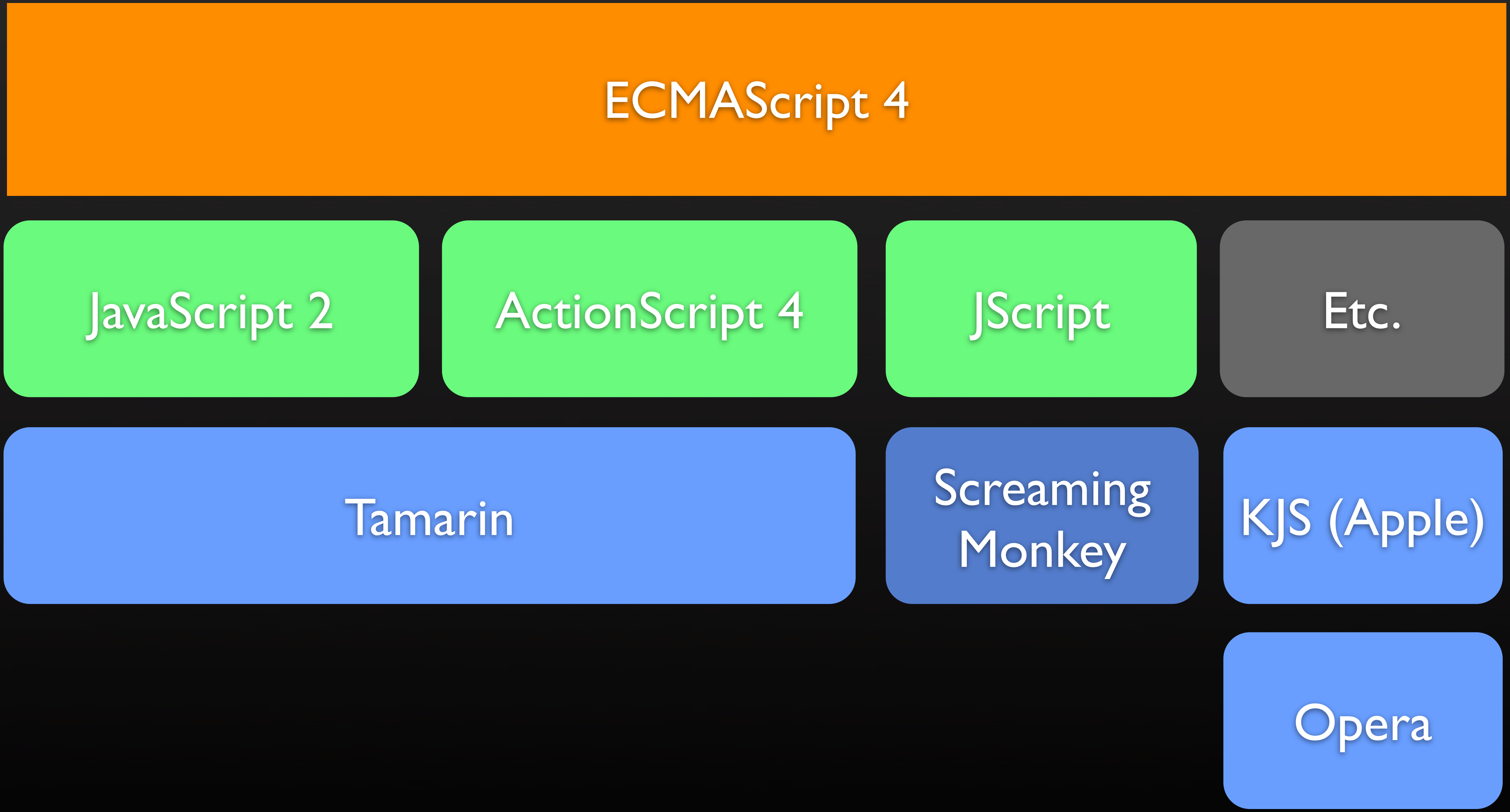
Tamarin and ECMAScript 4

John Resig (ejohn.org)
Mozilla Corporation

The Big Picture



The Direction



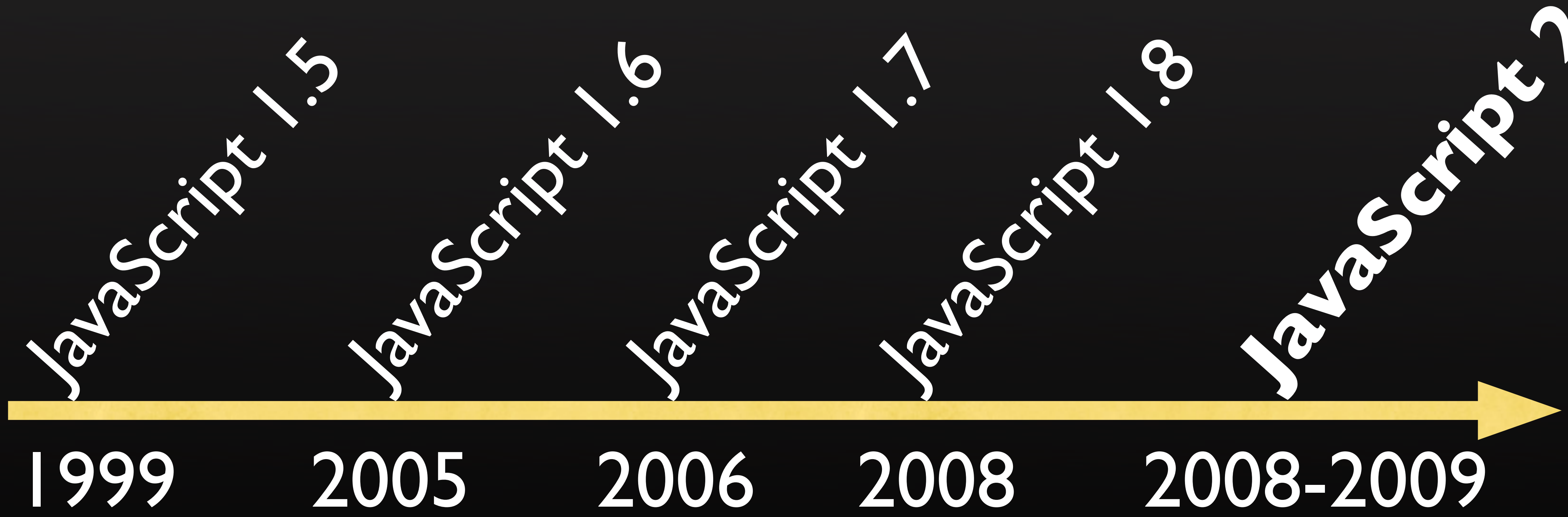
Tamarin

- Tamarin
  - New Virtual Machine from Adobe
  - Perfect for ActionScript
    - (a mutant cousin of JavaScript 2)
- The Three Monkeys:
  - ActionMonkey
  - ScreamingMonkey
  - IronMonkey

Three Monkies

- ActionMonkey
  - Integrating Tamarin into SpiderMonkey
  - Powering Firefox 4 (?) + JavaScript 2
- ScreamingMonkey
  - Bringing Tamarin into Internet Explorer
  - (Kicking and screaming?)
- IronMonkey
  - Bringing Python + Ruby to Tamarin

Path to JavaScript 2



The JavaScript Language

- Current:
  JavaScript 1.5 (ECMAScript 3)
- JavaScript 1.6 (Firefox 1.5)
- JavaScript 1.7 (Firefox 2)
- JavaScipt 1.8 (Firefox 3)
- ...
- **JavaScript 2** (ECMAScript 4)

ECMAScript 4 Goals

- Compatible with ECMAScript 3
- Suitable to developing large systems
- Allow for reusable libraries
- Merge previous efforts (ActionScript)
- Fix ECMAScript 3 bugs
- Keep it usable for small programs

Features

- Classes and Interfaces
- Packages and Namespaces
- Type Annotations
- Strict Verification
- Optimization
- Syntax Shortcuts
- Iterators and Generators
- Self-hosting

Classes

Classes

-
  ```
  class Programmer {
      var name;
      var city = “Boston, MA”;
      const interest = “computers”;
      function work() {}
  }
  ```

-
  ```
  var p = new Programmer;
  p.name = “John”;
  p.work();
  p.work.apply( someotherp );
  p.interest = “science”; // Error
  ```

Dynamic Classes

-
  ```
  dynamic class Programmer {
      var name;
      var city = “Boston, MA”;
      const interest = “computers”;
      function work() {}
  }
  ```

-
  ```
  var p = new Programmer;
  p.lastName = “Resig”;
  for ( var i in p )
      alert( i );
  // alert( “Resig” );
  ```

Getters and Setters

-
  ```
  class Programmer {
      var _name;
      function get name(){ return _name; }
      function set name(value){
          _name = value + " Resig";
      }
  }
  ```

-
  ```
  var p = new Programmer;
  p.name = "John";
  alert( p.name );
  // "John Resig"
  ```

Catch-Alls

```
dynamic class Programmer {
    meta function get(name) { … }
    meta function set(name, value) {
        alert("Setting " + name + " to " + value);
    }
}
```

```
var p = new Programmer
p.name = "John";
// alert("Setting name to John");
```

Inheritance

-
  ```
  class Artist {
      function draw() { alert(“Drawing!”); }
  }
  class Designer extends Artist {
      override function draw() {
          alert(“Designing!”);
      }
  }
  ```

-
  ```
  var d = new Designer
  d.draw();
  // alert(“Designing!”);
  ```

Inheritance (cont.)

- ‘final’ methods can’t be overriden

-
  ```
  class Artist {
      final function draw() {alert(“Drawing!”);}
  }
  class Designer extends Artist {
      // ERROR: Can’t override draw!
      override function draw() {
          alert(“Designing!”);
      }
  }
  ```

Inheritance (cont.)

- ‘final’ classes can’t be inherited from
- final class Artist {

```
final class Artist {
    function draw() { alert(“Drawing!”); }
}

// ERROR: Can’t inherit from Artist
class Designer extends Artist {
    override function draw() {
        alert(“Designing!”);
    }
}
```

Metaclass

- Provide global functions and properties on a class object

-
  ```
  class Users {
      static function find( name ) {
          // ...
      }
  }
  ```
- `Users.find( “John” );`

Interfaces

- Verify that a class implements another

-
  ```
  interface Artist {
      function draw();
  }
  class Designer implements Artist {
      function draw() { alert("Designing!"); }
  }
  ```

-
  ```
  var d = new Designer();
  if ( d is Artist )
     alert("Designers are Artists!");
  ```

Types

Numbers

- Numbers are now broken down into:
  - byte
  - int
  - uint
  - double (ECMAScript 3-style Number)
  - decimal

Type Annotations

- var name : string = “John”;
- let x : double = 5.3;
- function stuff(x: int, obj: Object) : boolean {}

Function Types

- Only return specific types:
```
function isValid() : boolean { }
```

- Only be used on certain objects types:
```
function every( this: Array, value: int ) {
    for ( var i = 0; i < this.length; i++ )
        alert( this[i] );
}
every.call( [0,1,2], 3 );
// alert(0); alert(1); alert(2);
every.call({a: “b”}, 4);
// ERROR
```

Rest Arguments

-
  ```
  function stuff( name, ...values ){
      alert( values.length );
  }
  ```

-
  ```
  stuff( "John", 1, 2, 3, 4 );
  // alert( 4 );
  ```

-
  ```
  function stuff( name : string, ...values : [int] ) : void {
      alert( values.length );
  }
  ```

Union and Any Types

- var test : (string, int, double) = “test”;
  test = 3;
  test = false; // ERROR
- type AnyNumber = (byte, int, double, decimal, uint);
- var test : AnyNumber = 3
- These are equivalent:
  - var test : * = “test”;
  - var test = “test”;

Type Definitions

- type Point = { x: int, y: int };
- var p : Point = { x: 3, y: 24 };

Nullability

- Prevent variables from accepting null values
- var name : * = “John”;
- var name : String! = “John”;
  name = “Ted”;
  name = null; // ERROR
- function test(name: String?) {
      alert(name);
  }

Initialization

-
  ```
  class User {
      var name : string!; // Must be initialized
      var last : string!;
      function User( n, l ) : name = n, last = l {
          // ...
      }
  }
  ```

“like”

-
  ```
  type Point = { x: int, y: int };
  ```

-
  ```
  if ( { x: 3, y: 5 } like Point )
     alert( “That looks like a point to me!” );
  ```

-
  ```
  if ( !({ x: 3 } like Point) )
     alert( “Not a point!” );
  ```

-
  ```
  // Loop over array-like things:
  function every( a: like { length: uint } ) {
     for ( var i = 0; i < a.length; i++ )
        alert( a[i] );
  }
  ```

“wrap”

- Force a type if compatible one doesn’t exist
- type Point = { x: int, y: int };
  var p: wrap Point = { x: 3, y: 8 };
- var p: Point = { x: 3, y: 8 } wrap Point;
- var p: Point = { x: 3, y: 8 } : Point;

Parameterized Types

-
  ```
  var m: Map.<Object, string>;
  ```

-
  ```
  class Point.<T> {
      var x: T, y: T;
  }
  ```

-
  ```
  var p = new Point.<double>;
  p.x = 3.0;
  p.y = 5.0;
  ```

Structure

For .. Each

- For each loops through values

-
  ```
  let s = “”;
  for each ( let n in [“a”,”b”,”c”] )
      s += n;
  alert(s);
  // “abc”
  ```

let statements

-
  ```
  for ( let i = 0; i < a.length; i++ )
      alert( a[i] );
  // i is undefined
  ```
- Using block statements:
  ```
  {
      let x = 5;
      {
          let x = 6;
          alert( x ); // 6
      }
      alert( x ); // 5
  }
  ```

let (cont.)

- let expressions:
```
var a = 5;
var x = 10 + (let (a=3) a) + a*a;
// x == 19
```
- let blocks:
```
let ( a=3 ) {
   alert( a ); // 3
}
// a is undefined
```
- let a = function(){ };

Packages

-
  ```
  package simple.tracker {
      internal var count: int = 0;
      public function add(){
          return ++count;
      }
  }
  ```

-
  ```
  import simple.tracker.*
  alert( add() ); // alert("1")
  count // ERROR, undefined
  ```

Namespaces

- namespace extra = “extra”;
- Pre-defined namespaces:
  - __ES4__
    - intrinsic
    - iterator
    - meta
- import dojo.query;
  import jquery.query;
  dojo::query(“#foo”)
  jquery::query(“div > .foo”)

Namespaces (cont.)

-
  ```
  import dojo.query;
  import jquery.query;
  use namespace dojo;
  query("#foo") // using dojo

  use namespace jquery;
  query("div > .foo") // using jquery
  ```

Multimethods

-
  ```
  generic function intersect(s1, s2);
  generic function intersect(s1: Shape, s2: Shape ) {
      // ...
  }
  generic function intersect(s1: Rect, s2: Rect ) {
      // ...
  }
  ```

Program Units

-
  ```
  use unit jQuery "http://jquery.com/jQuery"
  import com.jquery.*;
  new jQuery();
  ```

-
  ```
  unit jQuery {
      use unit Selectors "lib/Selectors";
      package com.jquery {
          class jQuery {
              function find() : jQuery {}
          }
      }
  }
  ```

Operator Overloading

-
  ```
  class Complex! { … }

  generic intrinsic function +(a: Complex, b: Complex)
      new Complex( a.real + b.real, a.imag + b.imag )

  generic intrinsic function +(a: Complex, b: AnyNumber)
      a + Complex(b)

  generic intrinsic function +(a: AnyNumber, b: Complex)
      Complex(a) + b
  ```

Self-Hosting

Map.es

- The reference implementation’s classes are written in ECMAScript

-
  ```
  package
  {
      use namespace intrinsic;
      use default namespace public;
      intrinsic class Map.<K,V>
      {
          static const length = 2;
          function Map(equals=intrinsic::===, hashcode=intrinsic::hashcode)
              : equals = equals
              , hashcode = hashcode
              , element_count = 0
          {
          }
          // ...
  }
  ```

More Info

- ECMAScript site:
  http://ecmascript.org/
- ECMAScript 4 White Paper Overview:
  http://www.ecmascript.org/es4/spec/overview.pdf
- Blogging:
  http://ejohn.org/