ECMAScript "3.1"

Pratap Lakshman, ECMA TC39 Representative

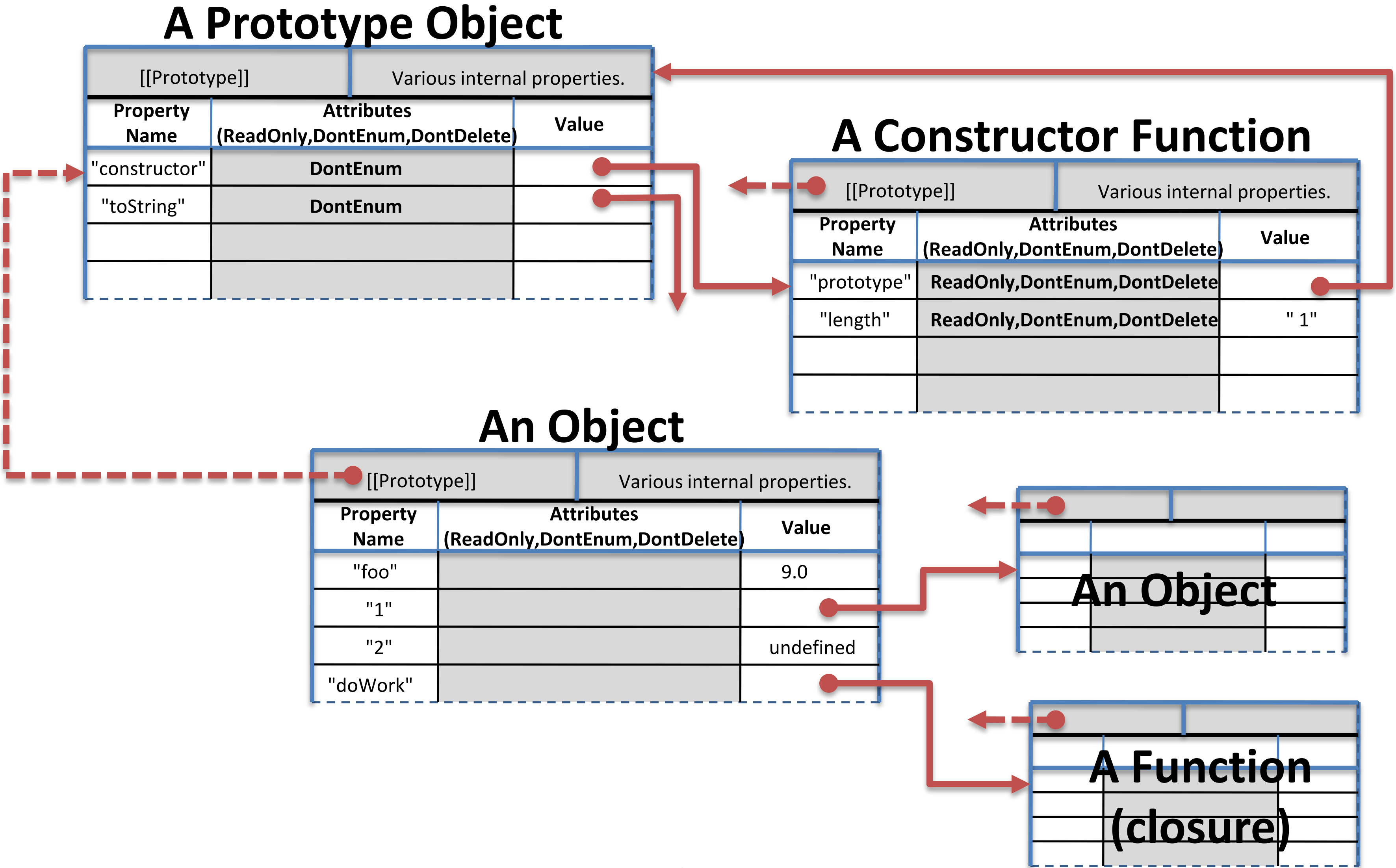
Microsoft Corp.

pratapL@microsoft.com

ECMAScript

- A brief introduction to the language
- Standardization
  - History, Motivation, and Status
- Evolving the Standard
  - Guiding Principles
- Two examples ECMAScript 3.1 features
  - Changes to the Object Model
  - "strict mode"
- Conclusion and Q&A

ECMAScript 3 Object Model



A Prototype Object

[[Prototype]]	Various internal properties.	
Property Name	Attributes (ReadOnly,DontEnum,DontDelete)	Value
"constructor"	DontEnum	
"toString"	DontEnum	

A Constructor Function

[[Prototype]]	Various internal properties.	
Property Name	Attributes (ReadOnly,DontEnum,DontDelete)	Value
"prototype"	ReadOnly,DontEnum,DontDelete	
"length"	ReadOnly,DontEnum,DontDelete	" 1"

An Object

[[Prototype]]	Various internal properties.	
Property Name	Attributes (ReadOnly,DontEnum,DontDelete)	Value
"foo"		9.0
"1"		
"2"		undefined
"doWork"		

ECMAScript 3 in syntax

```
var obj = new Object();
obj["foo"] = 9.0;                 // => obj.foo
obj["1"] = new String("hello");   // => obj[1]
```

Objects map strings to values

```
obj.doWork = function (n) { this.foo += n;}
obj.doWork(1); // => obj.foo == 10
```

Methods are function valued properties

```
function adder(x) {
  return function (y) { return y + x;}
}
```

Functions are first class objects

```
add2 = new adder(2);
add2(5);       // => 7
```

All functions can construct

```
function vehicle(kind) { this.kind = kind;}
vehicle.prototype.fuel = "petrol";
mycar = new vehicle("SUV"); // SUV running on petrol
mycar.fuel = "hybrid";      // override
```

All functions have a prototype property

Standardization
History, Motivation, and Status

- Standardized in 1997 as ECMA-262 Edition 1
- Revised in 1998 for ISO 16262 (Edition 2 Standard)
- Revised in 1999 with several feature additions (Edition 3 Standard, aka ES3)
- Ancillary specifications developed (E4X)

- In these 9 years, ES3 and the DOM have matured
- AJAX is driving use cases and dialects ahead of the standard
  - Need to resolve Interoperability concerns, harmonize divergences, and set the stage for the future

- Next revision, "ES3.1" in progress
  - Most of the work being done by "Working Groups"
    - All documents on the committee's wiki (http://wiki.ecmascript.org under the es3.1 namespace)
  - 2 browser based implementations before standardization
    - Standardization should not precede implementation
    - Demonstrate interoperability
  - Targeting ratification by June 2009

Evolving the Standard – Guiding Principles

- Ensure Stability and Interoperability
  - Don’t break my code!
    - "3 out of 4" rule for new syntax
  - Don’t even break the spec!
    - Using existing specification mechanisms (prose + algorithmic pseudo code)
    - Retaining existing section numbers, even
  - Codify proven non-standard extensions, and de-facto compatibility conventions
    - *Example:* Array "extras" from Mozilla
    - *Example:* JSON (based on the JSON2 reference implementation)
    - *Example:* Use previous reserved words as the names of object properties and to access them using "dot notation"
  - Bring specification closer to "reality"
    - *Example:* Fix the grammar for RegExp literals
  - Improve the language by reducing confusing or troublesome constructs
    - Easier said than done; can’t break existing code!
    - *Example:* "strict mode" opt-in for opting out of some features
    - *Example:* Augment Date to parse, and create, ISO date strings

Evolving the Standard – Guiding Principles

- Enable innovation
  - Be a friendly base for secure sub-languages
    - *Example:* no coercion of "this" to the global object in "strict mode"
    - *Example:* control the ReadOnly and DontDelete attributes on objects passed into secure sandboxes
  - Put language users on an equal footing with the language implementers
    - *Example:* Emulate standard built-in methods with regard to the DontEnum attribute
    - *Example:* convenience APIs to hook up, and look up, prototypes on objects.
  - Provide virtualizability, allowing for host object emulation
    - *Example:* Emulate host objects (like the DOM) through the programmatic creation and inspection of getter/setter properties
- Integrating features to work in combination is the key
  - And the most work
  - The whole is much more than the sum of its parts!

Changes to the Object Model

and

Object "meta" functions

Changes to the Object Model

- ES3 had only data properties
  - create -> set -> delete were the only state transitions possible
  - ReadOnly and DontDelete were "magic" attributes that controlled the latter two stages

- ES3.1 exposes these attributes programmatically, and reifies them
- … and introduces accessors (getter/setter) as a new kind of property
  - Subtle change to the semantics of the "set" and "delete" state transitions
    - Convert a data property to an accessor property or visa versa.
    - Change the state of a property attribute: writable, enumerable, configurable
    - Change/delete the getter and/or setter function of an accessor property.
    - Delete the property

- Rename/reinterpret attributes
  - [[ReadOnly]]--> [[Writable]]
  - [[DontEnum]] --> [[Enumerable]]
  - [[DontDelete]] --> [[Configurable]]

- If [[Configurable]] attribute is `false` for a property
  - None of the above can occur
  - [[Writable]] can be changed from `true` to `false`

Manipulating Properties and Attributes

```
Object.defineProperty(obj, propName, propDescriptor)
```

Functions on the Object constructor

Example:

```
Object.defineProperty(o, "length",
  {
    getter: function() { return this.computeLength(); },
    setter: function(value) { this.changeLength(value); }
  }
);

Object.defineProperty(o, "1",
  {
    value: 1, enumerable: true, configurable: true
  }
);
```

Define a property

```
Object.defineProperty(Array.prototype, "forEach",
  {
    enumerable: false, writable:false, configurable: false
  }
);
```

Modify property attributes

Retrieving a property description

```
Object.getOwnPropertyDescriptor(obj, propName);
```

Example:

```
var desc = Object.getOwnPropertyDescriptor(o, "length");

    desc => {
      getter: function() { return this.computeLength(); },
      setter: function(value) { this.changeLength(value); }
    }
```

- Return object is a descriptor with data properties

  `value, writable, enumerable, configurable`

  or

  `getter, setter, enumerable, configurable`

- Return object is usable as 3rd argument to `Object.defineProperty`

Object "lock down"

`Object.preventExtensions(obj)`

- Prevent adding properties to an object
  - [[Extensible]] property of Object set to `false`

`Object.seal(obj)`

- Prevent adding or reconfiguring properties
- Definitional structure of the object cannot be changed
  - State of the properties can still be modified, though
  - [[Configurable]] attribute of every owned property is set to `false`
  - [[Extensible]] property of Object set to `false`

`Object.freeze(obj)`

- Prevent adding, reconfiguring, modify the value of properties
  - Does what Object.seal does, in addition sets the [[Writable]] attribute of each own property to `false`
  - More aggressive form of lock down

These atomically place their object into the specified lock down state

Other Object "meta" functions

- Object.defineProperties(obj, descriptorSet)
- Object.create(protoObj, descriptorSet)
- Object.getOwnPropertyNames(obj)
- Object.getPrototypeOf(obj)
- Object.isExtensible(obj)
- Object.isSealed(obj)
- Object.isFrozen(obj)

Example

```
function point(x, y) {
 var self = Object.create(Point.prototype, {
   toString: {
    value: function () { return self.getX() + ' ' + self.getY();},
    enumerable: true
   },
   getX: {
    value: function () { return x;},
    enumerable: true
   },
   getY: {
    value: function () { return y;},
    enumerable: true
   }
  }
 );

 return self;
}
```

Example

```
function point(x, y) {
  var self = Object.create(Point.prototype, {
      toString: {
        value: Object.freeze(function () { return self.getX() + ' ' + self.getY();}),
        enumerable: true
      },
      getX: {
        value: Object.freeze(function () { return x;}),
        enumerable: true
      },
      getY: {
        value: Object.freeze(function () { return y;}),
        enumerable: true
      }
    }
  );

  return self;
}
```

"strict" mode

"strict" mode

- Secure Composition
  - When separately written programs are composed so that they may cooperate, how do you prevent them from interfering in unanticipated ways?

- ES3 has been too permissive
  - global scope and ambient reachability
  - "this" binding
  - with
  - eval
  - arguments aliasing

- But, cannot prohibit these without breaking existing code!

- "strict mode"
  - pragma to optionally constrain the syntax and semantics of the language

Areas of Language change

Areas of Language change ("approved in principle")

- The ability for a programmer to opt-in to using a "strict" subset of the language that performs additional error checks and restricts use of some error prone or insecure features of "ES3"
- Built-in support for JSON
- Getter/Setter properties with both syntactic support in Object literals and programmatic support via new property definition functions
- The ability to query and set property attributes such as “ReadOnly" and “DontDelete"
- The ability to "seal" objects in order to prevent modifications or additions to their properties
- Ability to query the prototype of an object
- Ability to create an object with a specified prototype

Areas of Language change ("approved in principle")

- The ability to use previous reserved words as the names of object properties and to access them using "dot notation"
- Array-like indexing for access to individual string characters
- The "array extra" functions first introduced in Mozilla's implementations and subsequently widely copied by others
- Support for parsing and creating ISO format date strings
- Ability to query the declared name of a function
- The ability to bind the "this" value or "arguments" of a function object to specific values
- Numerous specification bug fixes and clarifications intended to improve interoperability among implementations
- …

References

- ECMA-262 3rd Edition
  - http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
- ECMA TC39 wiki
  - http://wiki.ecmascript.org
  - All proposals for ECMAScript "3.1"
  - All ES 3.1 specification drafts
- JScript Deviations Document (available on the wiki)
  - Illustrates implementation drift since the Edition 3 Standard
- ECMAScript 3.1 Static Object Functions: Use Cases and Rationale (available on the wiki)
- JSON data interchance format RFC 4627
  - http://www.ietf.org/rfc/rfc4627.txt?number=4627
  - JSON2 reference implementation (http://www.json.org/json2.js)
- IEEE P754 - 2008 Standard for Binary Floating-Point Arithmetic (in publication process)

Summary and Conclusion

- Implementation convergence
  - Standardize proven de-facto extensions
  - Improve the specification, fix errors, and bring it closer to reality
- Make the language more expressive by providing greater access to the object model
  - Reducing "magic"
  - Enable high integrity programming
  - Liberate the ecosystem to innovate

Words of wisdom from R5RS

"Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. …"