Frontend Knowledge

JavaScript

Object Orientation, Inheritance & Prototype Chain

- Source: MDN
- JavaScript only knows one construct: `Object`
- Each object has a link to a prototype object

Properties

```
// parent object
var parent = {b: 3, c: 4};

// child object with inheritance
var child = Object.create(parent);
child.a = 1;
child.b = 2;

// prototype chain:
// child.[[Prototype]] = {b: 3, c: 4}
// child.__proto__ is deprecated
// since ES6 [[Prototype]] is accessed using Object.getPrototypeOf() and Object.setPrototypeOf()
// {a: 1, b: 2} » {b: 3, c: 4} » null

console.log(child.a); // 1
// Is there an 'a' own property on child? Yes, and its value is 1.

console.log(child.b); // 2
// Is there a 'b' own property on child? Yes, and its value is 2.
// The prototype also has a 'b' property, but it’s not visited.
// This is called "property shadowing"

console.log(child.c); // 4
// Is there a 'c' own property on child? No, check its prototype.
// Is there a 'c' own property on child.[[Prototype]]? Yes, its value is 4.

console.log(child.d); // undefined
// Is there a 'd' own property on child? No, check its prototype.
// Is there a 'd' own property on child.[[Prototype]]? No, check its prototype.
// child.[[Prototype]].[[Prototype]] is null, stop searching.
// no property found, return undefined
```

- Using getter and setter

```
// define a getter and setter for the year property
var d = Date.prototype;
Object.defineProperty(d, 'year', {
  get: function() { return this.getFullYear(); },
  set: function(y) { this.setFullYear(y); }
});

// use the getter and setter in a "Date" object
var now = new Date();
console.log(now.year); // 2016
new.year = 2015; // 2015
console.log(now); // Tue Aug 11 2015 11:23:16 GMT+0200 (CEST)
```

Methods

- Any function can be added to an object in the form of a property
- An inherited function acts just as any other property, including property shadowing (*method overriding*)
- When an inherited function is executed, the value of `this` points to the inheriting object, not to the prototype object where the function is an own property

```
// define object with property a and method m
var o = {
  a: 2,
  m: function(b) {
    return this.a + 1;
  }
};

console.log(o.m()); // 3
// When calling o.m in this case, "this" refers to o

var p = Object.create(o);
// p is an object that inherits from o

p.a = 4; // creates an own property "a" on p
console.log(p.m()); // 5
// When p.m is called, "this" refers to p
// So when p inherits the function m of o, "this.a" means p.a, the own
property "a" of p
```

Creating objects

Created with syntax constructs

```
var o = {a: 1};
// The newly created object o has Object.prototype as its [[Prototype]]
// o has no own property named "hasOwnProperty"
// hasOwnProperty is an own property of Object.prototype
// So o inherits hasOwnProperty from Object.prototype
// Object.prototype has null as its prototype
// o » Object.prototype » null

var a = ['yo', 'whadup', '?'];
// Arrays inherit from Array.prototype (which has methods like indexOf, forEach, etc.)
// The prototype chain looks like
// a » Array.prototype » Object.prototype » null

function f() {
  return 2;
}
// Functions inherit from Function.prototype (which has methods like call, bind, etc.)
// f » Function.prototype » Object.prototype » null
```

Created with a constructor

```
// A "constructor" in JavaScript is "just" a function that happens to be called with the new operator

function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype = {
  addVertex: function(v) {
    this.vertices.push(v);
  }
};

var g = new Graph();
// g is an object with own properties "vertices" and "edges"
// g.[[Prototype]] is the value of Graph.prototype when new Graph() is executed
```

Created with `Object.create`

```
// ES5 introduced a new method: Object.create()
// Calling this method creates a new object; prototype of this object is the first argument of the function

var a = {a: 1};
// a » Object.prototype » null

var b = Object.create(a);
// b » a » Object.prototype » null
console.log(b.a); // 1 (inherited)

var c = Object.create(b);
// c » b » a » Object.prototype » null

var d = Object.create(null);
// d » null
console.log(d.hasOwnProperty); // undefined, because d doesn't inherit from Object.prototype
```

Created with `class` keyword

```
// ES6 introduced a new set of keywords implementing classes (remaining prototype-based): class, constructor, static, extends, super

'use strict';

class Polygon {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}

class Square extends Polygon {
  constructor(sideLength) {
    super(sideLength, sideLength);
  }
  get area() {
    return this.height * this.width;
  }
  set sideLength(newLength) {
    this.height = newLength;
    this.width = newLength;
  }
}

var square = new Square(2);

console.log(square.area); // 4

square.sideLength = 3;
console.log(square.area); // 9
```

Performance

- Lookup for properties that are high up on the chain can have negative impact on performance
- Trying to access nonexisting properties will always traverse the full prototype chain
- When iterating over the properties of an object, **every** enumerable property that is on the prototype chain will be enumerated
- To check existence of property on own object use `hasOwnProperty`; inherited from `Object.prototype` (only thing in JS which deals with properties and does **not** traverse the prototype chain)

Bad Practice

- Don't extend `Object.prototype` or one of the other built-in prototypes (*monkey patching*) as it breaks *encapsulation*
- Only good reason is backporting newer JavaScript engine features; for example `Array.forEach`, etc.

Prototype Chain

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

A.prototype = {
  // Optimize speed by initializing instance variables
  varA: null,
  doSomething: function() {
    // ...
  }
}

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

B.prototype = Object.create(A.prototype, {
  varB: {
    value: null,
    enumerable: true,
    configurable: true,
    writable: true
  },
  doSomething: {
    // override
    value: function() {
      // call super
      A.prototype.doSomething.apply(this, arguments);
    },
    enumerable: true,
    configurable: true,
    writable: true
  }
});

B.prototype.constructor = B;

var b = new B();
b.doSomething();
```

- Important parts: Types are defined in `.prototype`, you use `Object.create()` to inherit
- Reference to the prototype object is copied to the internal `[[Prototype]]` property of the new instance
- When you access properties of the instance, JavaScript first checks object, and if not, it looks in `[[Prototype]]`

- This means that all the stuff you define in `prototype` is effectively shared by all instances
- You can even later change parts of `prototype` and have the changes appear in all existing instances

```
var a1 = new A();
var a2 = new A();
// Object.getPrototypeOf(a1).doSomething =
// Object.getPrototypeOf(a2).doSomething =
// A.prototype.doSomething
```

- `prototype` is for types, while `Object.getPrototypeOf()` is the same for instances
- `[[Prototype]]` is looked at *recursively*

```
var o = new Foo();

// JavaScript actually just does
var o = new Object();
o.[[Prototype]] = Foo.prototype;
Foo.call(o);
```

Hoisting

- Source: MDN
- Scope of a variable declared with `var` is its current *execution context* (enclosing function or global)
- Assigning a value to an undeclared variable implicitly creates it as a global variable
- Variable declarations are processed before any code is executed
- Variable can appear to be used before it's declared
- **Hoisting**: Variable declaration is moved to the top of the function or global code

ES5 Strict Mode

- Source: MDN
- A way to *opt in* to a restricted variant of JavaScript
- Eliminates some silent errors by changing them to throw errors
  - Impossible to accidentally create global variables
  - Makes assigments which would otherwise silently fail throw an exception
  - Throws an error if you attempt to delete undeletable properties
  - Requires that all properties named in an object literal be unique
  - Requires that function parameter names be unique
  - Forbids octal syntax
  - Forbids setting properties on primitive values
- Improves possibilities to perform optimizations by Engines (faster)
  - Prohibits `with`
  - `eval` of strict mode code does not introduce new variables into the surrounding

scope
    - Forbids deleting plain names: `var a; delete a;`
    - Names `eval` and `arguments` can't be bound or assigned
    - Doesn't alias properties of `arguments` object created within it
    - `arguments.callee`, `arguments.caller` and `caller` are no longer supported
    - value passed as `this` to a function is not forced into being an object (a.k.a *boxing*): primitive values are returned with their value, not as objects
- Prohibits some syntax likely to be defined in future versions of ES
    - List of identifiers become reserved keywords: `implements`, `interface`, `let`, `package`, `private`, `protected`, `public`, `static`, and `yield`
    - Prohibits function statements not at the top level of a script or function

Event Capturing & Bubbling

- Sources: MDN, Kirupa
- Every event starts at the root of the document, makes its way through the DOM and stops at the element that triggered the event (Event Capturing Phase)
- Once the event reaches its target, the event returns back to the root (Event Bubbling Phase)

```
// listen for click event during capturing phase
item.addEventListener('click', doSomething, true);

// listen for click event during bubbling phase
item.addEventListener('click', doSomething, false);

// listen for click, defaults to bubbling phase
item.addEventListener('click', doSomething);
```

- Call `stopPropagation()` on `Event` object to prevent it to be propagated further down or up
- Call `preventDefault()` to turn off default behavior of an element getting an event

Immediatly-invoded function expression (IIFE)

- Source: Ben Alman
- Every function, when invoked, creates a new execution context
- Invoking a function provides a very easy way to create privacy

```
(function() {
  // ...
})();
```

- Any function defined inside another function can access the outer function's passed-in arguments and variables (this relationship is known as a closure)
- IIFE can be used to „lock in" values and save state

```
var elems = document.getElementsByTagName('a');

// this doesn't work, because the value of "i" never gets locked in
// instead every link click alerts the total number of elements
for (var i=0; i<elems.length; i++) {
  elems[i].addEventListener('click', function(e) {
    e.preventDefault();
    alert('I am link #' + i);
  });
}

// this works, because inside the IIFE, the value of "i" is locked in as "lockedinIndex"
for (var i=0; i<elems.length; i++) {
  (function(lockedInIndex) {
    elems[i].addEventListener('click', function(e) {
      e.preventDefault();
      alert('I am link #' + lockedInIndex);
    });
  })(i);
}

// alternative
for (var i=0; i<elems.length; i++) {
  elems[i].addEventListener('click', (function(lockedInIndex) {
    return function(e) {
      e.preventDefault();
      alert('I am link #' + lockedInIndex);
    };
  })(i));
}
```

Web Components

- Sources: MDN Web Components, MDN Custom Elements, MDN HTML Templates, MDN Shadow DOM, MDN HTML Imports
- Web Components are reusable user interface widgets that are created using open Web technology
- Consists of four technologies: Custom Elements, HTML Templates, Shadow DOM, and HTML Imports

Custom Elements

- Capability for creating custom HTML tags and elements with own scripted behavior and CSS styling
- Attach behaviors to different parts of element's lifecycle
- Lifecycle callbacks
    - `constructor`: The behavior occurs when the element is created or upgraded

- `connectedCallback`: Called when the element is inserted into the DOM
- `disconnectedCallback`: Called when the element is removed from the DOM
- `attributeChangedCallback(attrName, oldVal, newVal)`: The behavior occurs when an attribute of the element is added, changed, or removed, including when these values are initially set

```
<flag-icon country="nl"></flag-icon>
```

```
class FlagIcon extends HTMLElement {

  constructor() {
    super();
    this._countryCode = null;
  }

  static get observedAttributes() {
    return ['country'];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    // name will always be "country" due to observedAttributes
    this._countryCode = newValue;
    this._updateRendering();
  }

  connectedCallback() {
    this._updateRendering();
  }

  get country() {
    return this._countryCode;
  }

  set country(v) {
    this.setAttribute('country', v);
  }

  _updateRendering() {
    // ...
  }

}

// Define element
customElements.define('flag-icon', FlagIcon);
```

HTML Templates

- HTML template element `<template>` is a mechanism for holding un-rendered client-side content
- Content fragment that is being stored for subsequent use
- Parser checks validity of content only

```
<table id="product-table">
  <thead>
    <tr>
      <th>UPC Code</th>
      <th>Product Name</th>
    </tr>
  </thead>
  <tbody>
  </tbody>
</table>

<template id="product-row">
  <tr>
    <td class="record"></td>
    <td></td>
  </tr>
</template>
```

Shadow DOM

- Provides encapsulation for the JavaScript, CSS, and templating in a Web Component
- Seperation from DOM
- Must always be attached to an existing element (literal element, or an element created by scripting): native or custom element

```
<html>
  <head></head>
  <body>
    <p id="hostElement"></p>
    <script>
      // create shadow DOM on the <p> element above
      var shadow = document.querySelector('#hostElement').createShadowRoot();
      // add some text to shadow DOM
      shadow.innerHTML = '<p>Here is some new text</p>';
      // add some css to make the text red
      shadow.innerHTML += '<style>p { color: red; }</style>';
    </script>
  </body>
</html>
```

HTML Imports

- Intended to be the packaging mechanism for Web Components

- Import an HTML file by using a `<link>` tag in an HTML document

```
<link rel="import" href="myfile.html">
```

Web Worker

- Source: MDN

Web Workers API

- Worker is an object (`new Worker()`) that runs a named JavaScript file
- Code runs in worker thread with another global context (no access to `window`)
- Dedicated worker is only accessible from the script that first spawned it, whereas shared workers can be accessed from multiple scripts
- Can't directly manipulate the DOM
- Data is sent between workers and the main thread via a system of messages (`postMessage()` method and `onmessage` event handler)
- Workers may spawn new workers (within same origin)

Dedicated workers

```
var first = document.querySelector('#number1');
var second = document.querySelector('#number2');
var result = document.querySelector('#result');

// Check if browser supports the Worker API
if (window.Worker) {
  var myWorker = new Worker('worker.js');

  var changeHandler = function() {
    myWorker.postMessage([first.value, second.value]);
  };

  first.onchange = changeHandler;
  second.onchange = changeHandler;

  myWorker.onmessage = function(e) {
    result.textContent = e.data;
  };
}
```

```
onmessage = function(e) {
  var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
  postMessage(workerResult);
}
```

- Immediately terminate a running worker from the main thread: `myWorker.terminate();`
- Workers may close themselves: `close();`

- When a runtime error occurs in the worker, its `onerror` event handler is called
- Have access to a global function, `importScripts()`

Shared workers

```
// multiply.js
var first = document.querySelector('#number1');
var second = document.querySelector('#number2');
var result = document.querySelector('#result');

if (!!window.SharedWorker) {
  var myWorker = new SharedWorker('worker.js');

  var changeHandler = function() {
    myWorker.postMessage([first.value, second.value]);
  };

  // ...
}
```

```
// square.js
var squareNumber = document.querySelector('#number3');
var result2 = document.querySelector('#result2');

if (!!window.SharedWorker) {
  var myWorker = new SharedWorker('worker.js');

  var changeHandler = function() {
    myWorker.postMessage([squareNumber.value, squareNumber.value]);
  };

  // ...
}
```

```
// worker.js
onconnect = function(e) {
  var port = e.ports[0];

  port.onmessage = function(e) {
    var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
    postMessage(workerResult);
  }
}
```

Web Applications & Frameworks

- Sources: Noeticforce, colorlib

- Interesting (beside Angular and React): Polymer, Riot

Ember

- Source: About
- Auto-updating Handlebars Templates: Ember makes Handlebars templates even better, by ensuring HTML stays up-to-date when the underlaying model changes
- Components: Create application-specific HTML tags, using Handlebars to describe their markup and JS to implement custom behavior
- Loading data from a server: Eliminates the boilerplate of displaying JSON retrieved from server
- Routing: Downright simple to create sophisticated, multi-page JS applications with great URL support

Aurelia

- Source: Features
- Forward-Thinking: Written with ES 2016; integrates Web Components
- Two-way databinding: Enables powerful two-way binding to any object by using adaptive techniques (efficient way to observe each property in model and automatically sync UI)
- Routing & UI composition: Pluggable pipeline, dynamic route patterns, child routers and asynchronous screen activation
- Broad language support: ES5, ES 2015 (ES6), ES 2016 (ES.Next) and TypeScript
- Modern architecture: Composed of smaller, focused modules
- Extensible HTML: Custom HTML elements, add custom attributes to existing elements and control template generation
- MV* with Conventions: Levery conventions to make constructing effortless
- Testable: ES 2015 modules combined with DI container make it easy to create highly cohesive, yet minimally coupled code, making unit testing a snap

Meteor

- Source: Introducing
- Full-stack JavaScript platform for developing modern web and mobile applications
- Includes a key set of technologies for building connected-client reactive applications, a build tool, and a curated set of packages
- Allows you to develop in one language (application server, web browser, and mobile device)
- Uses data on the wire, meaning the server sends data, not HTML, and the client renders it
- Embraces the ecosystem, bringing the best parts of the community in a careful and considered way
- Provides full stack reactivity, allowing UI to seamlessly reflect the true state with minimal development effort

Backbone

- Source: Getting Started
- Represent data as models, which can be created, validated, destroyed, and saved to the server
- Whenever a UI action causes an attribute of the model to change, the model triggers a „change“ event
- All views that display the model’s state can be notified of the change (able to respond

accordingly, re-rendering themselves with the new information)
- Minimal set of data-structuring (models and collections) and user interface (views and URLs)
- Helps to keep business logic separate from user interface

Polymer

- Source: Feature Overview
- Provides a set of features for creating custom elements
- Designed to make it easier and faster to make custom elements
- Elements can be instantiated (Constructor or `document.createElement`)
- Elements can be configured using attributes or properties
- Elements can be populated with internal DOM inside each instance
- Elements are responsive to property and attribute changes
- Elements are styled with internal defaults or externally
- Elements are responsive to methods that manipulate their internal state
- Features are divided into
    - Registration and lifecycle: Registering an element associated a class (prototype) with a custom element name. The element provides callbacks to manage its lifecycle. Use behaviors to share code
    - Declared properties: Declared properties can be configured from markup using attributes. Declared properties can optionally support change observers, two-way data binding, and reflection to attributes. You can also declare computed properties and read-only properties
    - Local DOM: Local DOM is the DOM created and managed by the element
    - Events: Attaching event listeners to the host object and local DOM children. Event retargeting.
    - Data binding: Property bindings. Binding to attributes.
    - Behaviors: Behaviors are reusable modules of code that can be mixed into Polymer elements.
    - Utility function: Helper methods for common tasks.
    - Experimental features and elements: Experimental template and styling features. Feature layering.

Knockout

- Source: Key concepts
- Declarative Bindings: Easily associate DOM elements with model data using a concise, readable syntax
- Automatic UI Refresh: When data model's state changes, UI updates automatically
- Dependency Tracking: Implicitly set up chains of relationships between model data, to transform and combine it
- Templating: Quickly generate sophisticated nested UIs as a function of model data

Vue

- Source: Overview
- Library for building interactive web interfaces
- Provide benefits of reactive data binding and composable view components
- Focused on view layer only
- Very easy to pick up and to integrate with other libraries or existing projects
- Embraces the concept of data-driven view (bind the DOM to the underlaying data)
- Small, self-contained, and often reusable components (very similar to Custom Elements)

Mercury

- Source: Mercury vs React
- Leverages Virtual DOM (immutable vdom structure)
- Comes with `observ-struct` (immutable data for state atom)
- Truly modular (swap out subsets)
- Encourages zero DOM manipulation
- Strongly encourages FRP (Functional reactive programming) techniquices and discourages local mutable state
- Highly performant (faster than React, Om, ember)

MobX

- Source: Concepts & Principles
- State: data that drives application (*domain specific state* like a list of todo items and *view state* such as the currently selected element)
- Derivations: *Computed values* (Derived from the current observable state using a pure function) and *Reactions* (Side effects that need to happen automatically if the state changes)
- Actions: Any piece of code that changes the state
- Supports an uni-directional data flow where *Actions* changes the *state*, which in turn updates all affected *views*
- Derivations are updated automatically, atomically and synchronously, computed values are updated lazily and should be pure (not supposed to change *state*)

Omniscient

- Source: Rationale
- Functional programming for UIs
- Memoization for stateless React components
- Top-down rendering of components (unidirectional data flow)
- Favors immutable data
- Encourages small, composable components, and shared functionality through mixins
- Natural separation of concern (components only deal with their own piece of data)
- Efficient (centrally defined `shouldComponentUpdate`)

Ractive.js

- Source: Ractive
- Live, reactive templating: Template-driven UI library
- Powerful and extensible: Two-way binding, animations, SVG support
- Optimised for your sanity: Ractive works for you and plays well with other libraries

WebRx

- Source: WebRx
- MVVM: Clean separation of concerns between View-Layer and Application-Layer by combining obversable View-Models with Two-Way declarative Data-Binding
- Components and Modules: Combine View-Models and View-Templates into self-contained, reusable chunks and package them into modules
- Client-Side Routing: Organize the parts into a state machine that maps Components onto pre-defined (optionally nested) regions of the page

Deku

- Source: Deku
- Library for rendering interfaces using pure functions and virtual DOM
- Pushed responsibility of all state management and side-effect onto tools like Redux
- Can be used in place of libraries like React and works well with Redux

Riot

- Source: Riot
- Brings Custom Tags to all browsers
- Human-readable
- Virtual DOM: Smallest possible amount of DOM updates and reflows, one way data flow, pre-compiled and cached expressions, lifecycle events for more control, server-side rendering for universal apps
- Close to standards
- Tooling friendly
- Think React + Polymer but without the bloat

Mithril

- Source: Mithril
- Client-side MVC framework
- Light-weight: small size, small API, small learning curve
- Robust: Safe-by-default templates, hierarchical MVC via components
- Fast: Virtual DOM diffing and compilable templates, intelligent auto-redrawing system

Stapes.js

- Source: Stapes.js
- Agnostic about your setup and style of coding
- Class creation, custom events, and data methods

Om

- Source: Om
- Global state management facilities built in
- Components may have arbitrary data dependencies, not limited to props & state
- Component construction can be intercepted via `:instrument` (simplifies debugging components and generic editors)
- Provides stream of all application state change deltas via `:tx-listen` (simplifies synchronization online and offline)
- Customizable semantics: Fine grained control over how components store state

Single Page Applications

Definition

- Goal: Provide a more fluid user experience
- Resources are dynamically loaded and added to the page as necessary, usually in response to user actions
- Page does not reload at any point in the process, nor does control transfer to another page

Pros

- More fluid user experience
- Mobile phone friendly

Cons

- Search engine optimization (lack of JavaScript execution on crawlers)
- Client/Server code partitioning (duplication of business logic)
- Browser history (breaks page history navigation using the Forward/Back buttons)
- Analytics (full page loads are required)
- Speed of initial load (slower first page load)
- JavaScript has to be enabled

ES5

- Sources: MDN, oio
- ECMAScript 5.0 released in 2009, ES 5.1 released in 2011 (maintenance)
- Introduced Strict mode
- Native JSON support: `JSON.parse()`, `JSON.stringify()`
- Property descriptor maps: Specify how the properties of your object can be altered after creation
  - `value`: The intrinsic value of the property
  - `writable`: Can value be changed after being set?
  - `enumerable`: Can property be iterated on for example in for-loops
  - `configurable`: Specifys if a property can be deleted and how the values of its property descriptor map can be modified

```
var obj = {};
Object.defineProperty(obj, 'attr', {
  value: 1,
  writable: true,
  enumerable: true,
  configurable: true
});
```

- Getters and Setters

```
var obj = {};
(function() {
  var _value = 1;
  Object.defineProperty(obj, 'value', {
    get: function() {
      return _value;
    },
    set: function(newValue) {
      _value = newValue;
    }
  });
})();

console.log(obj.value); // 1
obj.value = 5;
console.log(obj.value); // 5
```

Object

- `Object` constructor creates an object wrapper for the given value (empty object if value is `null` or `undefined`)
- `Object.assign()`: Creates a new object by copying the values of all enumerable own properties from one or more source objects to a target object
- `Object.create()`: Creates a new object with the specified prototype object and properties
- `Object.defineProperty()` / `Object.defineProperties()`: Adds the named property described by a given descriptor to an object
- `Object.entries()`: Returns an array of a given object's own enumerable property `[key, value]` pairs
- `Object.freeze()`: Freezes an object (can't delete or change any properties)
- `Object.getOwnPropertyDescriptor()` / `Object.getOwnPropertyDescriptors()`: Returns a property descriptor for a named property on an object or an array of all own
- `Object.getOwnPropertyNames()`: Returns an array containing the names of all of the given object's own properties
- `Object.getOwnPropertySymbols()`: Returns an array of all symbol properties found directly upon a given object
- `Object.getPrototypeOf()`: Returns the prototype of the specified object
- `Object.is()`: Compares if two values are the same
- `Object.isExtensible()`: Determines if extending of an object is allowed
- `Object.isFrozen()`: Determines if an object was frozen
- `Object.isSealed()`: Determines if an object is sealed
- `Object.keys()`: Returns an array containing the names of the given object's own enumerable properties
- `Object.preventExtensions()`: Prevents any extensions of an object
- `Object.seal()`: Prevents other code from deleting properties of an object
- `Object.setPrototypeOf()`: Sets the prototype (the internal `[[Prototype]]` property)

- `Object.values()`: Returns an array of a given object's own enumerable values

Array

- `Array.isArray(someVar)`: Check if `someVar` is an array
- `forEach()`: iterate on an array
- `map()`: iterate on an array and returns a new array with applied callback method
- `filter()`: iterate on an array and return a new array with elements which return true in callback method
- `every()`: returns `true` if callback method returns `true` for all elements
- `some()`: returns `true` if callback method returns `true` for at least one element
- `reduce()`: invokes callback method on every element and returns a single element

Function

- `Function.prototype.apply()`: Calls a function and sets its `this` to the provided value, arguments can be passed as an `Array` object
- `Function.prototype.bind()`: Creates a new function which, when called, has its `this` set to the provided value, with a given sequence of arguments preceding any provided when the new function was called
- `Function.prototype.call()`: Calls a function and sets its `this` to the provided value, arguments can be passed as they are

ES6

- Source: Luke Hoban

Arrows

- Function shorthand using `=>` syntax
- Support both statement block bodies as well as expression bodies which return value of expression
- Unlike functions, arrows share the same lexical `this` as their surrounding code

```
// Expression bodies
var odds = evens.map(v => v + 1);
var nums = evens.map((v, i) => v + i);
var pairs = evens.map(v => ({even: v, odd: v + 1}));

// Statement bodies
nums.forEach(v => {
  if (v % 5 === 0)
    fives.push(v);
});

// Lexical this
var bob = {
  _name: 'Bob',
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + ' knows ' + f)
    );
  }
}
```

Classes

- Sugar over the prototype-based OO pattern

```
class SkinnedMesh extends THREE.Mesh {

  constructor(geometry, materials) {
    super(geometry, materials);

    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
  }

  update(camera) {
    // ...
    super.update();
  }

  get boneCount() {
    return this.bones.length;
  }

  set matrixType(matrixType) {
    this.idMatrix = SkinnedMesh[matrixType]();
  }

  static defaultMatrix() {
    return new THREE.Matrix4();
  }

}
```

Enhanced Object Literals

- Support for setting the prototype at construction, shorthand for `foo: foo` assignments, defining methods, making super calls, and computing property names with expressions

```
var obj = {
  // __proto__
  __proto__: theProtoObj,
  // shorthand for "handler: handler"
  handler,
  // methods
  toString() {
    // super callse
    return 'd' + super.toString();
  },
  // computed (dynamic) property names
  [ 'prop_' + (() => 42) ]: 42
}
```

Template Strings

- Provide syntactic sugar for constructing strings

```
// Basic literal string creation
`In Javascript '\n' is a line-feed.`

// Multiline strings
`In Javascript this is
not legal.`

// String interpolation
var name = 'Bob', time = 'today';
`Hello ${name}, how are you ${time}?`

// Construct an HTTP request
// prefix is used to interpret the replacements and construction
POST`http://foo.org/bar?a=${a}&b=${b}
Content-Type: application/json
X-Credentials: ${credentials}
{ "foo": ${foo},
  "bar": ${bar}}`(myOnReadyStateChangeHandler);
```

Destructuring

- Makes it possible to extract data from array or objects into distinct variables

```
// syntax
var a, b, rest;
[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2

[a, b, ...rest] = [1, 2, 3, 4, 5];
console.log(a); // 1
console.log(b); // 2
console.log(rest); // [3, 4, 5]

({a, b} = {a: 1, b: 2});
console.log(a); // 1
console.log(b); // 2

// Array destructing ==
// default values
[a = 5, b = 7] = [1];
console.log(a); // 1
console.log(b); // 7

// swapping variables
```

```
var a = 1;
var b = 3;
[a, b] = [b, a];
console.log(a); // 3
console.log(b); // 1

// parsing an array returned from a function
function f() {
  return [1, 2];
}

var [a, b] = f();
console.log(a); // 1
console.log(b); // 2

// ignore some returned values
function f() {
  return [1, 2, 3];
}

var [a, , b] = f();
console.log(a); // 1
console.log(b); // 3

// Object destructing ==
// basic assignment
var o = {p: 42, q: true};
var {p, q} = o;

console.log(p); // 42
console.log(q); // true

// assigment without declaration
var a, b;
({a, b} = {a: 1, b: 2});

// assigning to new variable names
var {p: foo, q: bar} = o;

console.log(foo); // 42
console.log(bar); // true

// default values
var {a = 10, b = 5} = {a: 3};

console.log(a); // 3
console.log(b); // 5
```

Default + Rest + Spread

- Callee-evaluated default parameter values
- Turn an array into consecutive arguments in a function call
- Bind trailing paramters to an array

```
// default
function f(x, y=12) {
  return x + y;
}
console.log(f(3)); // 15

// rest
function f(x, ...y) {
  return x * y.length;
}
console.log(f(3, 'hello', true)); // 6

// spread: pass each element of array as argument
function f(x, y, z) {
  return x + y + z;
}
console.log(f(...[1, 2, 3])); // 6
```

Let + Const

- Block-scoped binding constructs
- `let` is the new var
- `const` is a single-assignment
- Static restrictions prevent use before assignment

```
function f() {
  {
    let x;
    {
      // okay, block scoped name
      const x = 'sneaky';
      // error, const
      x = 'foo';
    }
    // error, already declared in block
    let x = 'inner';
  }
}
```

Iterators + For..Of

- Enable custom iteration
- Generalize `for..in` to custom iterator-based iteration with `for..of`

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return {done: false, value: cur};
      }
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Generators

- Simplify iterator-authoring using `function*` and `yield`
- Function delcared as `function*` returns a Generator instance
- Generators are subtypes of iterators which include additional `next` and `throw`

```
var fibonacci = {
  [Symbol.iterator]: function*() {
    var pre = 0, cur = 1;
    for (;;) {
      var temp = pre;
      pre = cur;
      cur += temp;
      yield cur;
    }
  }
}

for (var n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Unicode

- Non-breaking additions to support full Unicode

Modules

- Language-level support for modules for component definition
- Codifies patterns from popular module loaders (AMD, CommonJS)

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;

// app.js
import * as math from 'lib/math';
alert('2π = ' + math.sum(math.pi, math.pi));

// otherApp.js
import {sum, pi} from 'lib/math';
alert('2π = ' + sum(pi, pi));

// lib/mathplusplus.js
export * from 'lib/math';
export var e = 2.71828182846;
export default function(x) {
  return Math.log(x);
}

// app.js
import ln, {pi, e} from 'lib/mathplusplus.js';
alert('2π = ' + ln(e) * pi * 2);
```

Module Loaders

- Support dynamic loading, state isolation, global namespace isolation, compilation hooks, nested virtualization
- Default loader can be configured
- New loaders can be constructed to evaluate and load code in isolated or constrained contexts

```
// dynamic loading - "System" is default loader
System.import('lib/math').then(function(m) {
  alert('2π = ' + m.sum(m.pi, m.pi));
});

// create execution sandboxes - new loaders
var loader = new Loader({
  global: fixup(window)
});
loader.eval('console.log("Hello, World!");');

// directly manipulate module cache
System.get('jquery');
System.get('jquery', Module({$: $}));
```

Map + Set + WeakMap + WeakSet

- Efficient data structures for common algorithms
- WeakMaps provides leak-free object-key'd side tables

```
// sets
var s = new Set();
s.add('hello').add('goodbye').add('hello');
s.size === 2;
s.has('hello') === true;

// maps
var m = new Map();
m.set('hello', 42);
m.set(s, 34);
m.get(s) === 34;

// weak maps
var wm = new WeakMap();
wm.set(s, {extra: 42});
wm.size === undefined;

// weak set
var ws = new WeakSet();
ws.add({data: 42});
```

Proxies

- Is used to define custom behavior for fundamental operations (e.g. property lookup, assignment, enumeration, function invocation, etc.)
- *handler*: Placeholder object which contains traps
- *traps*: The methods that provide property access
- *target*: Object which the proxy virtualizes

Basic example

```
var handler = {
  get: function(target, name) {
    return name in target ? target[name] : 37;
  }
};

var p = new Proxy({}, handler);
p.a = 1;
p.b = undefined;

console.log(p.a., p.b); // 1, undefined
console.log('c' in p, p.c); // false, 37
```

No-op forwarding proxy

```
var target = {};
var p = new Proxy(target, {});

p.a = 37; // operation forwarded to the target
console.log(target.a); // 37
```

Validation

```
let validator = {
  set: function(obj, prop, value) {
    if (prop === 'age') {
      if (!Number.isInteger(value)) {
        throw new TypeError('The age is not an integer');
      }
      if (value > 200) {
        throw new RangeError('The age seems invalid');
      }
    }

    obj[prop] = value;
  }
};

let person = new Proxy({}, validator);

person.age = 100;
console.log(person.age); // 100
person.age = 'young'; // Throws an exception
person.age = 300; // Throws an exception
```

Extending constructor

```
function extend(sup, base) {
  var descriptor = Object.getOwnPropertyDescriptor(base.prototype,
'constructor');
  base.prototype = Object.create(sup.prototype);
  var handler = {
    construct: function(target, args) {
      var obj = Object.create(base.prototype);
      this.apply(target, obj, args);
      return obj;
    },
    apply: function(target, that, args) {
      sup.apply(that, args);
      base.apply(that, args);
    }
  };
  var proxy = new Proxy(base, handler);
  descriptor.value = proxy;
  Object.defineProperty(base.prototype, 'constructor', descriptor);
  return proxy;
}

var Person = function(name) {
  this.name = name;
};

var Boy = extend(Person, function(name, age) {
  this.age = age;
});
Boy.prototype.sex = 'M';

var Peter = new Boy('Peter', 13);
console.log(Peter.sex); // 'M'
console.log(Peter.name); // 'Peter'
console.log(Peter.age); // 13
```

```
// proxying a normal object
var target = {};
var handler = {
  get: function(receiver, name) {
    return `Hello, ${name}!`;
  }
};

var p = new Proxy(target, handler);
p.world === 'Hello, world!';

// proxying a function object
var target = function() {
  return 'I am the target';
};
var handler = {
  apply: function(receiver, ...args) {
    return 'I am the proxy';
  }
};

var p = new Proxy(target, handler);
p() === 'I am the proxy';
```

Symbols

- Enable access control for object state
- Allow properties to be keyed by either `string` or `symbol`
- Symbols are a new primitive type

```
var MyClass = (function() {
  // module scoped symbol
  var key = Symbol('key');

  function MyClass(privateData) {
    this[key] = privateData;
  }

  MyClass.prototype = {
    doStuff: function() {
      ... this[key] ...
    }
  };

  return MyClass;
})();

var c = new MyClass('hello');
c['key'] === undefined
```

Subclassable Built-Ins

- Built-Ins can be subclassed

```
// Pseudo-code of Array
class Array {
  constructor(...args) { /* ... */ }
  static [Symbol.create]() {
    // ...
  }
}

// User code of Array subclass
class MyArray extends Array {
  constructor(...args) { super(args); }
}

// Two-phase "new":
// 1) Call @@create to allocate object
// 2) Invoke constructor on new instance
var arr = new MyArray();
arr[1] = 12;
arr.length == 2
```

Math + Number + String + Array + Object APIs

- Many new library additions, including core Math libraries, Array conversion helpers, String helpers, and Object.assign for copying

```
Number.EPSILON
Number.isInteger(Infinity) // false
Number.isNaN('NaN') // false

Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

'abcde'.includes('cd') // true
'abc'.repeat(3) // 'abcabcabc'

Array.from(document.querySelectorAll('*')) // returns a real array
Array.of(1, 2, 3) // similar to new Array(...), but without special one-arg behavior
[0, 0, 0].fill(7, 1) // [0, 7, 7]
[1, 2, 3].find(x => x == 3) // 3
[1, 2, 3].findIndex(x => x == 2) // 1
[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
['a', 'b', 'c'].entries() // iterator [0, 'a'], [1, 'b'], [2, 'c']
['a', 'b', 'c'].keys() // iterator 0, 1, 2
['a', 'b', 'c'].values() // iterator 'a', 'b', 'c'

Object.assign(Point, {origin: new Point(0, 0)});
```

Binary and Octal Literals

- Two new numeric literal forms are added for binary (`b`) and octal (`o`)

```
0b111110111 === 503 // true
0o767 === 503 // true
```

Promises

- Library for asynchronous programming
- First class representation of a value that may be made available in the future

```
function timeout(duration = 0) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, duration);
  });
}

var p = timeout(1000).then(() => {
  return timeout(2000);
}).then(() => {
  throw new Error('hmm');
}).catch(err => {
  return Promise.all([timeout(100), timeout(200)]);
});
```

TypeScript

- Source: TypeScript

Basic Types

```typescript
let isDone: boolean = false;

let decimal: number = 6;
let hex: number = 0xf00d;
let binary: number = 0b1010;
let octal: number = 0o744;

let color: string = 'blue';
let fullName: string = 'Bob Bobbington';
let sentence: string = `Hello, my name is ${fullName}.`

let list: number[] = [1, 2, 3];
let list: Array<number> = [1, 2, 3];

let x: [string, number];
x = ['hello', 10]; // correct
x = [10, 'hello']; // incorrect

enum Color {Red, Green, Blue};
let c: Color = Color.Green;

// start values at 1 instead of 0
enum Color {Red = 1, Green, Blue};
let c: Color = Color.Green;

// manually set the values
enum Color {Red = 1, Green = 2, Blue = 4};
let c: Color = Color.Green;
let colorName: string = Color[2]; // Green

let notSure: any = 4;
notSure = 'maybe a string instead';
notSure = false;

let list: any[] = [1, true, 'free'];

// no return type
function warnUser(): void {
  alert('This is a warning message');
}

// type assertions
let someValue: any = 'this is a string';
let strLength: number = (<string>someValue).length;
let strLength: number = (someValue as string).length;
```

Variable Declarations

- No use before declaration

- No re-declarations and Shadowing
- New scope per iteration if used in loops
- Support for Destructuring

```
// let
let hello = 'Hello!';

// block-scoping
function f(input: boolean) {
  let a = 100;

  if (input) {
    // still okay to reference 'a'
    let b = a + 1;
    return b;
  }

  // error: "b" doesn't exist here
  return b;
}

// const
const numLivesForCat = 9;
const kitty = {
  name: 'Aurora',
  numLives: numLivesForCat
};

// can’t re-assign to them
numLivesForCat = 8; // error

// but internal values are still modifiable
kitty.name = 'Rory';
kitty.numLives--;
```

Interfaces

- Duck Typing/Structural Subtyping: Interfaces fill the role of naming these types

```
// first interface
// parameter has to be an object and has to have a label property
function printLabel(labelledObj: {label: string}) {
  console.log(labelledObj.label);
}

let myObj = {size: 10, label: 'Size 10 Object'};
printLabel(myObj);

// with interface keyword
```

```ts
interface LabelledValue {
  label: string;
}

function printLabel(labelledObj: LabelledValue) {
  console.log(labelledObj.label);
}

// optional properties
interface SquareConfig {
  color?: string;
  width?: number;
}

function createSquare(config: SquareConfig): {color: string; area: number} {
  let newSquare = {color: 'white', area: 100};
  if (config.color) {
    newSquare.color = config.color;
  }
  if (config.width) {
    newSquare.area = config.width * config.width;
  }
  return newSquare;
}

let mySquare = createSquare({color: 'black'});

// additional properties
interface SquareConfig {
  color?: string;
  width?: number;
  [propName: string]: any;
}

// function types
interface SearchFunc {
  (source: string, subString: string): boolean;
}

let mySearch: SearchFunc;
mySearch = function(source: string, subString: string) {
  return source.search(subString) !== -1;
}

// indexable types
interface StringArray {
  [index: number]: string;
}

```

```
let myArray: StringArray;
myArray = ['Bob', 'Fred'];
let myStr: string = myArray[0];

// class types
interface ClockInterface {
  currentTime: Date;
  setTime(d: Date);
}

interface ClockConstructor {
  new (hour: number, minute: number);
}

class Clock implements ClockConstructor, ClockInterface {
  currentTime: Date;
  setTime(d: Date) {
    this.currentTime = d;
  }
  constructor(h: number, m: number) {}
}

// extending interfaces
interface Shape {
  color: string;
}

interface PenStroke {
  penWidth: number;
}

interface Square extends Shape, PenStroke {
  sideLength: number;
}

let square = <Square>{};
square.color = 'blue';
square.sideLength = 10;
square.penWidth: 5.0;

// hybrid types
interface Counter {
  (start: number): string;
  interval: number;
  reset(): void;
}

function getCounter(): Counter {
  let counter = <Counter>function (start: number) { };
```

```
  counter.interval = 123;
  counter.reset = function () { };
  return counter;
}

let c = getCounter();
c(10);
c.reset();
c.interval = 5.0;

// interfaces extending classes
class Control {
  private state: any
}

interface SelectableControl extends Control {
  select(): void;
}

class Button extends Control {
  select() { }
}
```

Classes

```
// classes
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return 'Hello, ' + this.greeting;
  }
}

let greeter = new Greeter('world');

// inheritance
class Animal {
  name: string;
  constructor(theName: string) { this.name = theName; }
  move(distanceInMeters: number = 0) {
    console.log(`${this.name} moved ${distanceInMeters}m.`);
  }
}

class Snake extends Animal {
  constructor(name: string) { super(name); }
```

```
  move(distanceInMeters = 5) {
    console.log('Slithering...');
    super.move(distanceInMeters);
  }
}

class Horse extends Animal {
  constructor(name: string) { super(name); }
  move(distanceInMeters = 45) {
    console.log('Galloping...');
    super.move(distanceInMeters);
  }
}

let sam = new Snake('Sammy the Python');
let tom: Animal = new Horse('Tommy the Palomino');

sam.move(); // Slithering... Sammy the Python moved 5m.
tom.move(34); // Galloping... Tommy the Palomino moved 34m.

// public, private, and protected modifiers
// public by default

// private
class Animal {
  private name: string;
  constructor(theName: string) { this.name = theName; }
}

new Animal('Cat').name; // error

// protected
class Person {
  protected name: string;
  constructor(name: string) { this.name = name; }
}

class Employee extends Person {
  private department: string;
  constructor(name: string, department: string) {
    super(name);
    this.department = department;
  }
  public getElevatorPitch() {
    return `Hello, my name is ${this.name} and I work in
${this.department}.`;
  }
}

```

```
let howard = new Employee('Howard', 'Sales');
console.log(howard.getElevatorPitch()); // Hello, my name is Howard and I work in Sales.
console.log(howard.name); // error

// parameter properties
class Animal {
  // shorthand to create and initialize the "name" member
  constructor(private name: string) { }
}

// static properties
// visible on the class itself rather than on the instances
class Grid {
  static origin = {x: 0, y: 0};
  calculateDistanceFromOrigin(point: {x: number, y: number}) {
    let xDist = (point.x - Grid.origin.x);
    let yDist = (point.y - Grid.origin.y);
    return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
  }
  constructor (public scale: number) { }
}

let grid1 = new Grid(1.0); // 1x scale
let grid2 = new Grid(5.0); // 5x scale

console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));

// abstract classes
abstract class Animal {
  // must be implemented in the derived classes
  abstract makeSound(): void;
  move(): void {
    console.log('roaming the earth...');
  }
}
```

Functions

```
// writing the function type
let myAdd: (x: number, y: number) => number =
  function(x: number, y: number): number { return x + y; }

// optional parameters
function buildName(firstName: string, lastName?: string) {
  return lastName? firstName + ' ' + lastName : lastName;
}

```

```
// default parameters
function buildName(firstName: string, lastName = 'Smith') {
  return firstName + ' ' + lastName;
}

// rest parameters
function buildName(firstName: string, ...restOfName: string[]) {
  return firstName + ' ' + restOfName.join(' ');
}

// lambdas and using "this"
let deck = {
  suits: ['hearts', 'spades', 'clubs', 'diamonds'],
  cards: Array(52),
  createCardPicker: function() {
    return function() {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);
      // "this" does reference "window" instead of "deck"
      return {
        suit: this.suits[pickedSuit],
        card: pickedCard % 13
      };
    };
  }
};

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert('card: ' + pickedCard.card + ' of ' + pickedCard.suit);

let deck = {
  suits: ['hearts', 'spades', 'clubs', 'diamonds'],
  cards: Array(52),
  createCardPicker: function() {
    // notice: the line below is now a lambda, allowing us to capture "this"
earlier
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);
      return {
        suit: this.suits[pickedSuit],
        card: pickedCard % 13
      };
    };
  }
};

```

```
// overloads
let suits = ['hearts', 'spades', 'clubs', 'diamonds'];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
  if (typeof x == 'object') {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  } else if (typeof x == 'number') {
    let pickedSuit = Math.floor(x / 13);
    return {suit: suits[pickedSuit], card: x % 13};
  }
}

let myDeck = [
  {suit: 'diamonds', card: 2},
  {suit: 'spades', card: 10},
  {suit: 'hearts', card: 4}
];
let pickedCard1 = myDeck[pickCard(myDeck)];
let pickedCard2 = pickCard(15);
```

Generics

Identity function example (think of it as `echo`)

```
// we loose type information hier
function identity(arg: any): any {
  return arg;
}

// use type variable
function identity<T>(arg: T): T {
  return arg;
}

// function declaration
let myIdentity: <T>(arg: T) => T = identity;

let output = identity<string>('myString'); // type of output will be "string"
// with type argument inference compiler will set value of "T"
let output = identity('myString'); // type of output will be "string"

// generic interface
interface GenericIdentityFn {
  <T>(arg: T): T;
}

```

```typescript
function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: GenericIdentityFn = identity;

// with type information
interface GenericIdentityFn<T> {
  (arg: T): T;
}

function identity<T>(arg: T): T {
  return arg;
}

let myIdentity: GenericIdentityFn<number> = identity;

// generic classes
class GenericNumber<T> {
  zeroValue: T;
  add: (x: T, y: T) => T;
}

let myGenericNumber = new GenericNumber<number>();
myGenericNumber.zeroValue = 0;
myGenericNumber.add = function(x, y) { return x + y; };

// generic constraints
interface Lengthwise {
  length: number;
}

// constraint is, that T has "length" member
function loggingIdentity<T extends Lengthwise>(arg: T): T {
  console.log(arg.length);
  return arg;
}

// using type parameters in generic constraints
function copyFields<T extends U, U>(target: T, source: U): T {
  for (let id in source) {
    target[id] = source[id];
  }
  return target;
}

let x = {a: 1, b: 2, c: 3, d: 4};

copyFields(x, {b: 10, d: 20}); // ok
```

```
copyFields(x, {Q: 90}); // error: property "Q" isn't declared in "x"
```

Enum

```
enum Direction {Up = 1, Down, Left, Right}

// reverse mapping
enum Enum {A}
let a = Enum.A; // 0
let nameOfA = Enum[Enum.A]; // "A"

// const enum
const enum Directions {Up, Down, Left, Right}
// generated code (without possibility to lookup names):
var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

Type Inference

```
let x = 3; // type is inferred to be "number"
let x = [0, 1, null]; // type is inferred with best common type algorithm
```

Type Compatibility

```
interface Named {
  name: string;
}

class Person {
  name: string;
}

let p: Named;
p = new Person(); // ok, because of structural typing

let x: Named;
// y's inferred type is {name: string, location: string}
let y: {name: 'Alice', location: 'Seattle'};
x = y;

let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // ok
x = y; // error
```

Symbols

```
let sym1 = Symbol();
// optional string key
let sym2 = Symbol('key');

// symbols are immutable and unique
let sym3 = Symbol('key');
sym2 === sym3; // false

// can be used as keys for object properties
let obj = {
  [sym]: 'value'
};
console.log(obj[sym]); // value
```

Iterators and Generators

```
// iterables
let someArray = [1, 'string', false];
for (let i in someArray) {
  console.log(i); // 0, 1, 2
}
for (let i of someArray) {
  console.log(i); // 1, "string", false
}
```

Namespaces

```
namespace Validation {
  export interface StringValidator {
    isAcceptable(s: string): boolean;
  }

  const lettersRegexp = /^[A-Za-z]+$/;
  const numberRegexp = /^[0-9]+$/;

  export class LettersOnlyValidator implements StringValidator {
    isAcceptable(s: string) {
      return lettersRegexp.test(s);
    }
  }

  export class ZipCodeValidator implements StringValidator {
    isAcceptable(s: string) {
      return numberRegexp.test(s);
    }
  }
}

```

```
let strings = ['Hello', '98052', '101'];

let validators: {[s: string]: Validation.StringValidator; } = {};
validators['ZIP code'] = new Validation.ZipCodeValidator();
validators['Letters only'] = new Validation.LettersOnlyValidator();

for (let s of strings) {
  for (let name in validators) {
    console.log(`"${s}" - ${validators[name].isAccetable(s) ? 'matches' :
'does not match'} ${name}`);
  }
}

// splitting across files
// validation.ts
namespace Validation {
  // ...
}

// letters-only-validator.ts
/// <reference path="validation.ts" />
namespace Validation {
  // ...
}

// aliases
namespace Shapes {
  export namespace Polygons {
    export class Triangle { }
    export class Square { }
  }
}

import polygons = Shapes.Polygons;
let sq = new polygons.Square();

// ambient namespaces
declare namespace D3 {
  export interface Selectors {
    select: {
      (selector: string): Selection;
      (element: EventTarget): Selection;
    }
  }

  export interface Event {
    x: number;
    y: number;
  }
```

```
70
71    export interface Base extends Selectors {
72      event: Event;
73    }
74  }
75
76  declare var d3: D3.Base;
```

Namespaces and Modules

```
1  // myModules.d.ts
2  declare module "SomeModule" {
3    export function fn(): string;
4  }
5
6  // myOtherModule.ts
7  /// <reference path="myModules.d.ts" />
8  import * as m from "SomeModule";
```

JSX

- Embeddable XML-like syntax
- Meant to be transformed into valid JavaScript
- In order to use JSX:
  1. Name files with a `.tsx` extension
  2. Enable the `jsx` option
- Angle bracket type assertions are disallowed in `.tsx` files: `var foo = bar as foo;` instead of `var foo = <foo>bar;`

Mixins

```
1  // disposable mixin
2  class Disposable {
3    isDisposed: boolean;
4    dispose() {
5      this.isDisposed = true;
6    }
7  }
8
9  // activatable mixin
10 class Activatable {
11   isActive: boolean;
12   activate() {
13     this.isActive = true;
14   }
15   deactivate() {
16     this.isActive = false;
```

```
  }
}

class SmartObject implements Disposable, Activatable {
  constructor() {
    setInterval(() => console.log(this.isActive + ' : ' + this.isDisposed),
500);
  }

  interact() {
    this.activate();
  }

  isDisposed: boolean = false;
  dispose: () => void;

  isActive: boolean = false;
  activate: () => void;
  deactivate: () => void;
}

function applyMixins(derivedCtor: any, baseCtors: any[]) {
  baseCtors.forEach(baseCtor => {
    Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
      derivedCtor.prototype[name] = baseCtor.prototype[name];
    });
  });
}

applyMixins(SmartObject, [Disposable, Activatable]);

let smartObj = new SmartObject();
setTimeout(() => smartObj.interact(), 1000);
```

Bundler

Webpack

- Source: Webpack
- Most pressing reason for development was Code Splitting and modularized static assets
- Goals:
  - Split dependency tree into chunks loaded on demand
  - Keep initial loading time low
  - Every static asset should be able to be a module
  - Ability to integrate 3rd-party libraries as modules
  - Ability to customize nearly every part of the module bundler
  - Suited for big projects

How is webpack different?

- Code Splitting
  - webpack has two types of dependencies in its tree: sync and async
  - async dependencies act as split points and form a new chunk
  - after chunk tree optimization a file for each chunk is emitted
- Loaders
  - used to transform other resources into JavaScript
  - by doing so, every resource forms a module
- Clever parsing
  - can nearly process every 3rd party library
  - handles most common module styles: CommonJS and AMD
- Plugin system
  - features a rich plugin system
  - most internal features are based on this
  - possibility to customize webpack

jspm

- Package manager for the SystemJS universal module loader, built on top of the dynamic ES6 module loader
- Loads any module format (ES6, AMD, CommonJS and globals) directly from any registry such as `npm` and `Github` with flat versioned dependency management
- For development: Load modules as separate files with ES6 and plugins compiled in the brwoser
- For production: Optimize into a bundle, layered bundles or a self-executing bundle with a single command

rollup

- JavaScript module bundler
- Allows writing application or library as a set of modules (using ES5 `import`/`export` syntax)
- Bundle them up into a single file
  - A bundle is more portable and easier to consume than a collection of files
  - Compression works better with fewer bigger files
  - In the browser, a 100kb bundle loads much faster than 5 20kb files (not valid for HTTP/2)
  - By bundling code, we can take advantage of tree-shaking (fewer wasted bytes)

Testing

Jasmine

- Source: Jasmine

- Behavior-driven development framework for testing
- Does not depend on any other JavaScript framework
- Does not require a DOM
- Could be run from command line with jasmine-node

Mocha

- Source: Mocha
- Feature-rich JavaScript test framework running on Node.js and in the browser
- There are different assertion libraries: Node.js' built-in assert module, should.js (BDD), expect.js, chai, better-assert, unexpected

Jest

- Source: Jest
- Uses Jasmine assertions by default
- Virtualizes JavaScript environments, provides browser mocks and runs test in parallel across workers
- Automatically mocks JavaScript modules, making most existing code testable

Other Tools

- Test Coverage: Istanbul
- Static Code Analysis: Sidekick
- Static Code Analysis: Plato
- Linting: eslint
- Web Performance Metrics Collector and Monitoring Tool: phantomas

AngularJS

What is it?

- Source: What is Angular 1?
- Structural framework for dynamic web apps
- HTML as template language with extended syntax
- Data binding & dependency injection
- Attempts to minimize the impedance mismatch between document centric HTML and what an application needs by creating new HTML constructs (*Directives*)
- Well-defined structure for all of the DOM and AJAX glue code
- Opinionated about how a CRUD application should be built
- Everything you need: Data-binding, basic templating directives, form validation, routing, deep-linking, reusable components, dependency injection
- Testability story: Unit-testing, end-to-end testing, mocks and test harnesses
- Seed application with directory layout and test scripts as a starting point

- Simplifies application development by presenting a higher level of abstraction (comes at a cost of flexibility)
- CRUD applications are a good fit, Games and GUI editors are not
- Belief that declarative code is better than imperative:
    - Decouple DOM manipulation from app logic (improves testability)
    - Regard app testing as equal in importance to app writing
    - Decouple client side of an app from the server side
    - Common tasks should be trivial and difficult tasks should be possible
- Angular frees you from the following pains:
    - Registering callbacks
    - Manipulating HTML DOM programmatically
    - Marshaling data to and from the UI
    - Writing tons of initialization code just to get started

Pros/Cons

Pros

- Quick prototyping
- Development is fast once you're familiar with it
- Very expressive (less code)
- Easy testability
- Good for apps with highly interactive client side code
- Two-way data binding
- Dependency injection system
- Extends HTML

Cons

- Learning curve becomes very steep
- Complexity of DI and services
- Scopes are easy to use, but hard to debug
- Documentation is definitely not up to par
- Directives are powerful, but difficult to use
- Lack of configuration after Bootstrap
- Router is limited
- Search engine indexability

Performance Issues

- Source: Performance
- Accessing the DOM is expensive
- Any time a new scope is created, that adds more values for the garbage collector
- Every scope stores an array of functions: `$$watchers`
- Every time `$watch` is called on a scope value, or a value is bound from the DOM a function gets added to the `$$watchers` array of the innermost scope
- When any value in scope changes, all watchers in the `$$watchers` array will fire, and if any

of them modify a watched value, they will all fire again (will continue until a full pass of the `$$watchers` array makes no changes)
- Use bind-once syntax where possible: `{{::scopeValue}}`
- `$on`, `$broadcast`, and `$emit` are slow as events have to walk entire scope hierarchy
- Always call `$on('$destroy')`

The bad parts

- ng-click and other DOM events
- scope.$watch
- scope.$on
- Directive postLink
- ng-repeat
- ng-show and ng-hide

The good (performant) parts

- track by
- oneTime bindings with ::
- compile and preLink
- $evalAsync (queue operations up for execution at the end of the current digest cycle)
- Services, scope inheritance, passing objects by reference
- $destroy
- unbinding watches and event listeners
- ng-if and ng-switch

Dependency Injection

- Source: DI
- Components such as services, directives, filters, and animations are defined by an injectable factory method or constructor function
- Controllers are defined by a constructor function, which can be injected with components as dependencies, but can also be provided with special dependencies
- The `run` method cannot inject providers
- The `config` method cannot inject services or values

Route Handling

- Source: Component Router
- Recommended to develop apps as a hierarchy of isolated components with own UI and well defined programmatic interface to the component that contains it
- Root Router matches it's *Route Config* against the URL; if a *Route Definition* in the *Route Config* recognizes a part of the URL then the *Component* associated with the *Route Definition* is instantiated and rendered in the *Outlet*
- If the new *Component* contains routes of its own then a new *Router* (Child Router) is created for this *Routing Component*

Comparison to Backbone and React

- Backbone: 3rd party templating (underscore), No two-way binding, Unopinionated
- React: No routing, Uni-directional data flow, Virtual DOM (faster updates), Probably used with flux (architecture template with dispatcher)

Angular CLI

- Source: CLI
- CLI for Angular 2 applications based on ember-cli
- Build system now uses Webpack as well

Angular Universal

- Source: Universal
- Server-side Rendering for Angular 2 apps
- Better perceived performance
- Optimized for Search Engines
- Site Preview

RxJS

- Source: RxJS
- Event-driven, resilient and responsive Architecture
- Set of libraries for composing asynchronous and event-based programs
- Developers represent asynchronous data streams with Observables, query asynchronous data streams using Operators, and parameterize the concurrency in async data streams using Schedulers
- Observable sequences are data streams

ngrx/store

- Source: Introduction
- RxJS powered state management inspired by Redux for Angular 2 apps
- Store builds on the concepts made popular by Redux (state management container for React) supercharged with the backing of RxJS
- Three main pieces: *Reducers*, *Actions*, and a single application *Store*
- Store (*Database*)
    - „Single source of truth“,
    - Snapshot of Store at any point will supply a complete representation of relevant application state
    - Centralized, immutable state
- Reducers (*Tables*)
    - A pure function, accepting two arguments, the previous state and an action with a type and optional data (payload) associated with the event
- Actions
    - All interaction that causes a state update

    - All relevant user events are dispatched as actions, flowing through the action pipeline defined by store
- Dispatch » Reducers » New State » Store

```
export const counter: Reducer<number> = (state: number = 0, action: Action) =>
{
  switch (action.type) {
    case 'INCREMENT':
      return state + 1;
    case 'DECREMENT':
      return state - 1;
    default:
      return state;
  }
};
```

ReactJS

What is it?

- Source: Why React?
- JavaScript library for creating user interfaces (the **V** in *MVC*)
- Simple: Express how your app should look at any given point in time, React will automatically manage all UI updates when your underlaying data changes
- Declarative: React conceptually hits the „refresh“ button, and knows to only update the changed parts
- Build composable components: With React the *only* you do is build encapsulated components (easier code reuse, testing and separation of concerns)

Pros/Cons

Pros

- Extremely easy to write UI test cases (due to virtual DOM system)
- Reusability of components (even combine them)
- Plays well together with other libraries or frameworks
- Automatic UI updates when underlaying data changes
- Ease of debugging (Chrome Extension)
- Works nicely with CommonJS/AMD patterns

Cons

- Learning curve for beginners
- Integrating into a traditional MVC framework like rails would require some configuration
- Kind of verbose (isn’t as straight forward as pure HTML & JS
- Not a full framework (no router nor model management)

Flux

- Source: Flux
- An application architecture for React utilizing a unidirectional data flow
- Three major parts: **Dispatcher**, **Stores** and **Views** (React components)
- When a user interacts with a React view, the view propagates an action through a central dispatcher, to the various stores that hold the application's data and business logic, which updates all of the views that are affected
- Control is inverted with stores: the stores accept updates and reconcile them as appropriate, rather than depending on something external to update its data in a consistent way
- Unidirectional data flow: dispatcher, stores and views are independent nodes with distinct inputs and outputs; action creators are simple, discrete, semantic helper function that facilitate passing data to the dispatcher in the form of an action

React Native

- Source: Tutorial
- Uses native components instead of web components as building blocks
- Real mobile apps are built – no mobile web apps
- Instead of recompiling you can reload your app instantly
- Use native code when you need to

Redux

- Source: Redux, Three Principles
- Predictable state container
- Single source of truth: The application state is stored in an object tree within a single store
- State is read-only: Only way to change the state is to emit an action, an object describing what happened
- Changes are made with pure function: Pure reducers specify how the state tree is transformed by actions