

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

- Each property is searched along the prototype chain

```

1 // parent object
2 var parent = {b: 3, c: 4};
3
4 // child object with inheritance
5 var child = Object.create(parent);
6 child.a = 1;
7 child.b = 2;
8
9 // prototype chain:
10 // child.[[Prototype]] = {b: 3, c: 4}
11 // child.__proto__ is deprecated
12 // since ES6 [[Prototype]] is accessed using Object.getPrototypeOf() and
   Object.setPrototypeOf()
13 // {a: 1, b: 2} » {b: 3, c: 4} » null
14
15 console.log(child.a); // 1
16 // Is there an 'a' own property on child? Yes, and its value is 1.
17
18 console.log(child.b); // 2
19 // Is there a 'b' own property on child? Yes, and its value is 2.
20 // The prototype also has a 'b' property, but it's not visited.
21 // This is called "property shadowing"
22
23 console.log(child.c); // 4
24 // Is there a 'c' own property on child? No, check its prototype.
25 // Is there a 'c' own property on child.[[Prototype]]? Yes, its value is 4.
26
27 console.log(child.d); // undefined
28 // Is there a 'd' own property on child? No, check its prototype.
29 // Is there a 'd' own property on child.[[Prototype]]? No, check its
   prototype.
30 // child.[[Prototype]].[[Prototype]] is null, stop searching.
31 // no property found, return undefined

```

- Using getter and setter

```

1 // define a getter and setter for the year property
2 var d = Date.prototype;
3 Object.defineProperty(d, 'year', {
4   get: function() { return this.getFullYear(); },
5   set: function(y) { this.setFullYear(y); }
6 });
7
8 // use the getter and setter in a "Date" object
9 var now = new Date();
10 console.log(now.year); // 2016
11 new.year = 2015; // 2015
12 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

```
1 // define object with property a and method m
2 var o = {
3   a: 2,
4   m: function(b) {
5     return this.a + 1;
6   }
7 };
8
9 console.log(o.m()); // 3
10 // When calling o.m in this case, "this" refers to o
11
12 var p = Object.create(o);
13 // p is an object that inherits from o
14
15 p.a = 4; // creates an own property "a" on p
16 console.log(p.m()); // 5
17 // When p.m is called, "this" refers to p
18 // So when p inherits the function m of o, "this.a" means p.a, the own
   property "a" of p
```

Creating objects

1. Created with syntax constructs

```

1  var o = {a: 1};
2  // The newly created object o has Object.prototype as its [[Prototype]]
3  // o has no own property named "hasOwnProperty"
4  // hasOwnProperty is an own property of Object.prototype
5  // So o inherits hasOwnProperty from Object.prototype
6  // Object.prototype has null as its prototype
7  // o » Object.prototype » null
8
9  var a = ['yo', 'whadup', '?'];
10 // Arrays inherit from Array.prototype (which has methods like indexOf,
    // forEach, etc.)
11 // The prototype chain looks like
12 // a » Array.prototype » Object.prototype » null
13
14 function f() {
15     return 2;
16 }
17 // Functions inherit from Function.prototype (which has methods like call,
    // bind, etc.)
18 // f » Function.prototype » Object.prototype » null

```

1. Created with a constructor

```

1  // A "constructor" in JavaScript is "just" a function that happens to be
    // called with the new operator
2
3  function Graph() {
4      this.vertices = [];
5      this.edges = [];
6  }
7
8  Graph.prototype = {
9      addVertex: function(v) {
10         this.vertices.push(v);
11     }
12 };
13
14 var g = new Graph();
15 // g is an object with own properties "vertices" and "edges"
16 // g.[[Prototype]] is the value of Graph.prototype when new Graph() is
    // executed

```

1. Created with `Object.create`

```
1 // ES5 introduced a new method: Object.create()
2 // Calling this method creates a new object; prototype of this object is the
  first argument of the function
3
4 var a = {a: 1};
5 // a » Object.prototype » null
6
7 var b = Object.create(a);
8 // b » a » Object.prototype » null
9 console.log(b.a); // 1 (inherited)
10
11 var c = Object.create(b);
12 // c » b » a » Object.prototype » null
13
14 var d = Object.create(null);
15 // d » null
16 console.log(d.hasOwnProperty()); // undefined, because d doesn't inherit from
  Object.prototype
```

1. Created with `class` keyword

```

1 // ES6 introduced a new set of keywords implementing classes (remaining
  // prototype-based): class, constructor, static, extends, super
2
3 'use strict';
4
5 class Polygon {
6   constructor(height, width) {
7     this.height = height;
8     this.width = width;
9   }
10 }
11
12 class Square extends Polygon {
13   constructor(sideLength) {
14     super(sideLength, sideLength);
15   }
16   get area() {
17     return this.height * this.width;
18   }
19   set sideLength(newLength) {
20     this.height = newLength;
21     this.width = newLength;
22   }
23 }
24
25 var square = new Square(2);
26
27 console.log(square.area); // 4
28
29 square.sideLength = 3;
30 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 nonexistent 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 thin 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

```
1 function A(a) {
2     this.varA = a;
3 }
4
5 A.prototype = {
6     // Optimize speed by initializing instance variables
7     varA: null,
8     doSomething: function() {
9         // ...
10    }
11 }
12
13 function B(a, b) {
14     A.call(this, a);
15     this.varB = b;
16 }
17
18 B.prototype = Object.create(A.prototype, {
19     varB: {
20         value: null,
21         enumerable: true,
22         configurable: true,
23         writable: true
24     },
25     doSomething: {
26         // override
27         value: function() {
28             // call super
29             A.prototype.doSomething.apply(this, arguments);
30         },
31         enumerable: true,
32         configurable: true,
33         writable: true
34     }
35 });
36
37 B.prototype.constructor = B;
38
39 var b = new B();
40 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

```

1 var a1 = new A();
2 var a2 = new A();
3 // Object.getPrototypeOf(a1).doSomething =
4 // Object.getPrototypeOf(a2).doSomething =
5 // A.prototype.doSomething

```

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

```

1 var o = new Foo();
2
3 // JavaScript actually just does
4 var o = new Object();
5 o.[[Prototype]] = Foo.prototype;
6 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 assignments 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)

```
1 // listen for click event during capturing phase
2 item.addEventListener('click', doSomething, true);
3
4 // listen for click event during bubbling phase
5 item.addEventListener('click', doSomething, false);
6
7 // listen for click, defaults to bubbling phase
8 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

Immediately-invoked 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

```
1 (function() {
2     // ...
3 })();
```

- 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

```

1  var elems = document.getElementsByTagName('a');
2
3  // this doesn't work, because the value of "i" never gets locked in
4  // instead every link click alerts the total number of elements
5  for (var i=0; i<elems.length; i++) {
6      elems[i].addEventListener('click', function(e) {
7          e.preventDefault();
8          alert('I am link #' + i);
9      });
10 }
11
12 // this works, because inside the IIFE, the value of "i" is locked in as
13 // "lockedInIndex"
14 for (var i=0; i<elems.length; i++) {
15     (function(lockedInIndex) {
16         elems[i].addEventListener('click', function(e) {
17             e.preventDefault();
18             alert('I am link #' + lockedInIndex);
19         });
20     })(i);
21 }
22 // alternative
23 for (var i=0; i<elems.length; i++) {
24     elems[i].addEventListener('click', (function(lockedInIndex) {
25         return function(e) {
26             e.preventDefault();
27             alert('I am link #' + lockedInIndex);
28         };
29     })(i));
30 }

```

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

Old example:

```
1 If nothing appeared below, then your browser is not supporting Custom Elements.  
2 <x-product data-name="Ruby" data-img="https://s3-us-west-  
  2.amazonaws.com/s.cdn.io/4621/ruby.png" data-url="http://example.com/1"></x-  
  product>  
3 <x-product data-name="JavaScript" data-img="https://s3-us-west-  
  2.amazonaws.com/s.cdn.io/4621/javascript.png" data-url="http://example.com/2">  
  </x-product>  
4 <x-product data-name="Python" data-img="https://s3-us-west-  
  2.amazonaws.com/s.cdn.io/4621/python.png" data-url="http://example.com/3"></x-  
  product>
```

```

1 // Create a new object based of the HTMLElement prototype
2 var XProductProto = Object.create(HTMLElement.prototype);
3
4 // Set up the element
5 XProductProto.createdCallback = function() {
6     // Create a Shadow Root
7     var shadow = this.createShadowRoot();
8
9     // Create a standard img element and set it's attributes
10    var img = document.createElement('img');
11    img.alt = this.getAttribute('data-name');
12    img.src = this.getAttribute('data-img');
13    img.width = '150';
14    img.height = '150';
15    img.className = 'product-img';
16
17    // Add the image to the Shadow Root
18    shadow.appendChild(img);
19
20    // Add an event listener to the image
21    img.addEventListener('click', function(e) {
22        window.location = this.getAttribute('data-url');
23    });
24
25    // Create a link to the product
26    var link = document.createElement('a');
27    link.innerText = this.getAttribute('data-name');
28    link.href = this.getAttribute('data-url');
29    link.className = 'product-name';
30
31    // Add the link to the Shadow Root
32    shadow.appendChild(link);
33 };
34
35 // Register the new element
36 var XProduct = document.registerElement('x-product', {
37     prototype: XProductProto
38 });

```

New example

```

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

```

```

1  class FlagIcon extends HTMLElement {
2
3      constructor() {
4          super();
5          this._countryCode = null;
6      }
7
8      static get observedAttributes() {
9          return ['country'];
10     }
11
12     attributeChangedCallback(name, oldValue, newValue) {
13         // name will always be "country" due to observedAttributes
14         this._countryCode = newValue;
15         this._updateRendering();
16     }
17
18     connectedCallback() {
19         this._updateRendering();
20     }
21
22     get country() {
23         return this._countryCode;
24     }
25
26     set country(v) {
27         this.setAttribute('country', v);
28     }
29
30     _updateRendering() {
31         // ...
32     }
33
34 }
35
36 // Define element
37 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

```

1 <table id="product-table">
2   <thead>
3     <tr>
4       <th>UPC Code</th>
5       <th>Product Name</th>
6     </tr>
7   </thead>
8   <tbody>
9   </tbody>
10 </table>
11
12 <template id="product-row">
13   <tr>
14     <td class="record"></td>
15     <td></td>
16   </tr>
17 </template>

```

Shadow DOM

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

```

1 <html>
2   <head></head>
3   <body>
4     <p id="hostElement"></p>
5     <script>
6       // create shadow DOM on the <p> element above
7       var shadow = document.querySelector('#hostElement').createShadowRoot();
8       // add some text to shadow DOM
9       shadow.innerHTML = '<p>Here is some new text</p>';
10      // add some css to make the text red
11      shadow.innerHTML += '<style>p { color: red; }</style>';
12    </script>
13  </body>
14 </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

```

1 <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

```
1 var first = document.querySelector('#number1');
2 var second = document.querySelector('#number2');
3 var result = document.querySelector('#result');
4
5 // Check if browser supports the Worker API
6 if (window.Worker) {
7     var myWorker = new Worker('worker.js');
8
9     var changeHandler = function() {
10         myWorker.postMessage([first.value, second.value]);
11     };
12
13     first.onchange = changeHandler;
14     second.onchange = changeHandler;
15
16     myWorker.onmessage = function(e) {
17         result.textContent = e.data;
18     };
19 }
```

```
1 onmessage = function(e) {
2     var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
3     postMessage(workerResult);
4 }
```

- 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

```

1 // multiply.js
2 var first = document.querySelector('#number1');
3 var second = document.querySelector('#number2');
4 var result = document.querySelector('#result');
5
6 if (!!window.SharedWorker) {
7     var myWorker = new SharedWorker('worker.js');
8
9     var changeHandler = function() {
10         myWorker.postMessage([first.value, second.value]);
11     };
12
13     // ...
14 }

```

```

1 // square.js
2 var squareNumber = document.querySelector('#number3');
3 var result2 = document.querySelector('#result2');
4
5 if (!!window.SharedWorker) {
6     var myWorker = new SharedWorker('worker.js');
7
8     var changeHandler = function() {
9         myWorker.postMessage([squareNumber.value, squareNumber.value]);
10     };
11
12     // ...
13 }

```

```

1 // worker.js
2 onconnect = function(e) {
3     var port = e.ports[0];
4
5     port.onmessage = function(e) {
6         var workerResult = 'Result: ' + (e.data[0] * e.data[1]);
7         postMessage(workerResult);
8     }
9 }

```

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 underlying 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, ES 2016 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: Leverage 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 underlying 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) techniques 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 observable 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`: Specifies if a property can be deleted and how the values of its property descriptor map can be modified

```
1 var obj = {};  
2 Object.defineProperty(obj, 'attr', {  
3   value: 1,  
4   writable: true,  
5   enumerable: true,  
6   configurable: true  
7 });
```

- Getters and Setters

```

1  var obj = {};
2  (function() {
3      var _value = 1;
4      Object.defineProperty(obj, 'value', {
5          get: function() {
6              return _value;
7          },
8          set: function(newValue) {
9              _value = newValue;
10         }
11     });
12 })();
13
14 console.log(obj.value); // 1
15 obj.value = 5;
16 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

```

1 // Expression bodies
2 var odds = evens.map(v => v + 1);
3 var nums = evens.map((v, i) => v + i);
4 var pairs = evens.map(v => ({even: v, odd: v + 1}));
5
6 // Statement bodies
7 nums.forEach(v => {
8   if (v % 5 === 0)
9     fives.push(v);
10 });
11
12 // Lexical this
13 var bob = {
14   _name: 'Bob',
15   _friends: [],
16   printFriends() {
17     this._friends.forEach(f =>
18       console.log(this._name + ' knows ' + f)
19     );
20   }
21 }

```

Classes

- Sugar over the prototype-based OO pattern


```

1  class SkinnedMesh extends THREE.Mesh {
2
3      constructor(geometry, materials) {
4          super(geometry, materials);
5
6          this.idMatrix = SkinnedMesh.defaultMatrix();
7          this.bones = [];
8          this.boneMatrices = [];
9      }
10
11     update(camera) {
12         // ...
13         super.update();
14     }
15
16     get boneCount() {
17         return this.bones.length;
18     }
19
20     set matrixType(matrixType) {
21         this.idMatrix = SkinnedMesh[matrixType]();
22     }
23
24     static defaultMatrix() {
25         return new THREE.Matrix4();
26     }
27
28 }

```

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

```

1  var obj = {
2      // __proto__
3      __proto__: theProtoObj,
4      // shorthand for "handler: handler"
5      handler,
6      // methods
7      toString() {
8          // super callse
9          return 'd' + super.toString();
10     },
11     // computed (dynamic) property names
12     [ 'prop_' + (() => 42) ]: 42
13 }

```

Template Strings

- Provide syntactic sugar for constructing strings

```
1 // Basic literal string creation
2 `In Javascript '\n' is a line-feed.`
3
4 // Multiline strings
5 `In Javascript this is
6 not legal.`
7
8 // String interpolation
9 var name = 'Bob', time = 'today';
10 `Hello ${name}, how are you ${time}?`
11
12 // Construct an HTTP request
13 // prefix is used to interpret the replacements and construction
14 POST`http://foo.org/bar?a=${a}&b=${b}
15 Content-Type: application/json
16 X-Credentials: ${credentials}
17 { "foo": ${foo},
18   "bar": ${bar}}`(myOnReadyStateChangeHandler);
```

Destructuring

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

```
1 // syntax
2 var a, b, rest;
3 [a, b] = [1, 2];
4 console.log(a); // 1
5 console.log(b); // 2
6
7 [a, b, ...rest] = [1, 2, 3, 4, 5];
8 console.log(a); // 1
9 console.log(b); // 2
10 console.log(rest); // [3, 4, 5]
11
12 ({a, b} = {a: 1, b: 2});
13 console.log(a); // 1
14 console.log(b); // 2
15
16 // Array destructing ==
17 // default values
18 [a = 5, b = 7] = [1];
19 console.log(a); // 1
20 console.log(b); // 7
21
22 // swapping variables
```

```

23 var a = 1;
24 var b = 3;
25 [a, b] = [b, a];
26 console.log(a); // 3
27 console.log(b); // 1
28
29 // parsing an array returned from a function
30 function f() {
31     return [1, 2];
32 }
33
34 var [a, b] = f();
35 console.log(a); // 1
36 console.log(b); // 2
37
38 // ignore some returned values
39 function f() {
40     return [1, 2, 3];
41 }
42
43 var [a, , b] = f();
44 console.log(a); // 1
45 console.log(b); // 3
46
47 // Object destructuring ==
48 // basic assignment
49 var o = {p: 42, q: true};
50 var {p, q} = o;
51
52 console.log(p); // 42
53 console.log(q); // true
54
55 // assignment without declaration
56 var a, b;
57 ({a, b} = {a: 1, b: 2});
58
59 // assigning to new variable names
60 var {p: foo, q: bar} = o;
61
62 console.log(foo); // 42
63 console.log(bar); // true
64
65 // default values
66 var {a = 10, b = 5} = {a: 3};
67
68 console.log(a); // 3
69 console.log(b); // 5

```

Default + Rest + Spread

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

```

1  // default
2  function f(x, y=12) {
3      return x + y;
4  }
5  console.log(f(3)); // 15
6
7  // rest
8  function f(x, ...y) {
9      return x * y.length;
10 }
11 console.log(f(3, 'hello', true)); // 6
12
13 // spread: pass each element of array as argument
14 function f(x, y, z) {
15     return x + y + z;
16 }
17 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

```

1  function f() {
2      {
3          let x;
4          {
5              // okay, block scoped name
6              const x = 'sneaky';
7              // error, const
8              x = 'foo';
9          }
10         // error, already declared in block
11         let x = 'inner';
12     }
13 }

```

Iterators + For..Of

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

```

1  let fibonacci = {
2    [Symbol.iterator]() {
3      let pre = 0, cur = 1;
4      return {
5        next() {
6          [pre, cur] = [cur, pre + cur];
7          return {done: false, value: cur};
8        }
9      }
10   }
11 }
12
13 for (var n of fibonacci) {
14   // truncate the sequence at 1000
15   if (n > 1000)
16     break;
17   console.log(n);
18 }

```

Generators

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

```

1  var fibonacci = {
2    [Symbol.iterator]: function*() {
3      var pre = 0, cur = 1;
4      for (;;) {
5        var temp = pre;
6        pre = cur;
7        cur += temp;
8        yield cur;
9      }
10   }
11 }
12
13 for (var n of fibonacci) {
14   // truncate the sequence at 1000
15   if (n > 1000)
16     break;
17   console.log(n);
18 }

```

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)

```
1 // lib/math.js
2 export function sum(x, y) {
3   return x + y;
4 }
5 export var pi = 3.141593;
6
7 // app.js
8 import * as math from 'lib/math';
9 alert('2π = ' + math.sum(math.pi, math.pi));
10
11 // otherApp.js
12 import {sum, pi} from 'lib/math';
13 alert('2π = ' + sum(pi, pi));
14
15 // lib/mathplusplus.js
16 export * from 'lib/math';
17 export var e = 2.71828182846;
18 export default function(x) {
19   return Math.log(x);
20 }
21
22 // app.js
23 import ln, {pi, e} from 'lib/mathplusplus.js';
24 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

```

1 // dynamic loading - "System" is default loader
2 System.import('lib/math').then(function(m) {
3     alert('2π = ' + m.sum(m.pi, m.pi));
4 });
5
6 // create execution sandboxes - new loaders
7 var loader = new Loader({
8     global: fixup(window)
9 });
10 loader.eval('console.log("Hello, World!");');
11
12 // directly manipulate module cache
13 System.get('jquery');
14 System.get('jquery', Module({$: $}));

```

Map + Set + WeakMap + WeakSet

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

```

1 // sets
2 var s = new Set();
3 s.add('hello').add('goodbye').add('hello');
4 s.size === 2;
5 s.has('hello') === true;
6
7 // maps
8 var m = new Map();
9 m.set('hello', 42);
10 m.set(s, 34);
11 m.get(s) === 34;
12
13 // weak maps
14 var wm = new WeakMap();
15 wm.set(s, {extra: 42});
16 wm.size === undefined;
17
18 // weak set
19 var ws = new WeakSet();
20 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

```
1 var handler = {
2   get: function(target, name) {
3     return name in target ? target[name] : 37;
4   }
5 };
6
7 var p = new Proxy({}, handler);
8 p.a = 1;
9 p.b = undefined;
10
11 console.log(p.a, p.b); // 1, undefined
12 console.log('c' in p, p.c); // false, 37
```

No-op forwarding proxy

```
1 var target = {};
2 var p = new Proxy(target, {});
3
4 p.a = 37; // operation forwarded to the target
5 console.log(target.a); // 37
```

Validation

```
1 let validator = {
2   set: function(obj, prop, value) {
3     if (prop === 'age') {
4       if (!Number.isInteger(value)) {
5         throw new TypeError('The age is not an integer');
6       }
7       if (value > 200) {
8         throw new RangeError('The age seems invalid');
9       }
10    }
11
12    obj[prop] = value;
13  }
14 };
15
16 let person = new Proxy({}, validator);
17
18 person.age = 100;
19 console.log(person.age); // 100
20 person.age = 'young'; // Throws an exception
21 person.age = 300; // Throws an exception
```

Extending constructor


```
1 function extend(sup, base) {
2   var descriptor = Object.getOwnPropertyDescriptor(base.prototype,
'constructor');
3   base.prototype = Object.create(sup.prototype);
4   var handler = {
5     construct: function(target, args) {
6       var obj = Object.create(base.prototype);
7       this.apply(target, obj, args);
8       return obj;
9     },
10    apply: function(target, that, args) {
11      sup.apply(that, args);
12      base.apply(that, args);
13    }
14  };
15  var proxy = new Proxy(base, handler);
16  descriptor.value = proxy;
17  Object.defineProperty(base.prototype, 'constructor', descriptor);
18  return proxy;
19 }
20
21 var Person = function(name) {
22   this.name = name;
23 };
24
25 var Boy = extend(Person, function(name, age) {
26   this.age = age;
27 });
28 Boy.prototype.sex = 'M';
29
30 var Peter = new Boy('Peter', 13);
31 console.log(Peter.sex); // 'M'
32 console.log(Peter.name); // 'Peter'
33 console.log(Peter.age); // 13
```

```

1  // proxying a normal object
2  var target = {};
3  var handler = {
4      get: function(receiver, name) {
5          return `Hello, ${name}!`;
6      }
7  };
8
9  var p = new Proxy(target, handler);
10 p.world === 'Hello, world!';
11
12 // proxying a function object
13 var target = function() {
14     return 'I am the target';
15 };
16 var handler = {
17     apply: function(receiver, ...args) {
18         return 'I am the proxy';
19     }
20 };
21
22 var p = new Proxy(target, handler);
23 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

```

1  var MyClass = (function() {
2      // module scoped symbol
3      var key = Symbol('key');
4
5      function MyClass(privateData) {
6          this[key] = privateData;
7      }
8
9      MyClass.prototype = {
10         doStuff: function() {
11             ... this[key] ...
12         }
13     };
14
15     return MyClass;
16 })();
17
18 var c = new MyClass('hello');
19 c['key'] === undefined

```

Subclassable Built-Ins

- Built-Ins can be subclassed

```

1  // Pseudo-code of Array
2  class Array {
3      constructor(...args) { /* ... */ }
4      static [Symbol.create]() {
5          // ...
6      }
7  }
8
9  // User code of Array subclass
10 class MyArray extends Array {
11     constructor(...args) { super(args); }
12 }
13
14 // Two-phase "new":
15 // 1) Call @@create to allocate object
16 // 2) Invoke constructor on new instance
17 var arr = new MyArray();
18 arr[1] = 12;
19 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

```

1  Number.EPSILON
2  Number.isInteger(Infinity) // false
3  Number.isNaN('NaN') // false
4
5  Math.acosh(3) // 1.762747174039086
6  Math.hypot(3, 4) // 5
7  Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2
8
9  'abcde'.includes('cd') // true
10 'abc'.repeat(3) // 'abcabcabc'
11
12 Array.from(document.querySelectorAll('*')) // returns a real array
13 Array.of(1, 2, 3) // similar to new Array(...), but without special one-arg
    behavior
14 [0, 0, 0].fill(7, 1) // [0, 7, 7]
15 [1, 2, 3].find(x => x == 3) // 3
16 [1, 2, 3].findIndex(x => x == 2) // 1
17 [1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
18 ['a', 'b', 'c'].entries() // iterator [0, 'a'], [1, 'b'], [2, 'c']
19 ['a', 'b', 'c'].keys() // iterator 0, 1, 2
20 ['a', 'b', 'c'].values() // iterator 'a', 'b', 'c'
21
22 Object.assign(Point, {origin: new Point(0, 0)});

```

Binary and Octal Literals

- Two new numeric literal forms are added for binary (`0b`) and octal (`0o`)

```

1  0b111110111 === 503 // true
2  0o767 === 503 // true

```

Promises

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

```
1 function timeout(duration = 0) {
2   return new Promise((resolve, reject) => {
3     setTimeout(resolve, duration);
4   });
5 }
6
7 var p = timeout(1000).then(() => {
8   return timeout(2000);
9 }).then(() => {
10   throw new Error('hmm');
11 }).catch(err => {
12   return Promise.all([timeout(100), timeout(200)]);
13 });
```

TypeScript

- Source: [TypeScript](#)

Basic Types

```

1  let isDone: boolean = false;
2
3  let decimal: number = 6;
4  let hex: number = 0xf00d;
5  let binary: number = 0b1010;
6  let octal: number = 0o744;
7
8  let color: string = 'blue';
9  let fullName: string = 'Bob Bobbington';
10 let sentence: string = `Hello, my name is ${fullName}.`
11
12 let list: number[] = [1, 2, 3];
13 let list: Array<number> = [1, 2, 3];
14
15 let x: [string, number];
16 x = ['hello', 10]; // correct
17 x = [10, 'hello']; // incorrect
18
19 enum Color {Red, Green, Blue};
20 let c: Color = Color.Green;
21
22 // start values at 1 instead of 0
23 enum Color {Red = 1, Green, Blue};
24 let c: Color = Color.Green;
25
26 // manually set the values
27 enum Color {Red = 1, Green = 2, Blue = 4};
28 let c: Color = Color.Green;
29 let colorName: string = Color[2]; // Green
30
31 let notSure: any = 4;
32 notSure = 'maybe a string instead';
33 notSure = false;
34
35 let list: any[] = [1, true, 'free'];
36
37 // no return type
38 function warnUser(): void {
39     alert('This is a warning message');
40 }
41
42 // type assertions
43 let someValue: any = 'this is a string';
44 let strLength: number = (<string>someValue).length;
45 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

```
1 // let
2 let hello = 'Hello!';
3
4 // block-scoping
5 function f(input: boolean) {
6   let a = 100;
7
8   if (input) {
9     // still okay to reference 'a'
10    let b = a + 1;
11    return b;
12  }
13
14  // error: "b" doesn't exist here
15  return b;
16 }
17
18 // const
19 const numLivesForCat = 9;
20 const kitty = {
21   name: 'Aurora',
22   numLives: numLivesForCat
23 };
24
25 // can't re-assign to them
26 numLivesForCat = 8; // error
27
28 // but internal values are still modifiable
29 kitty.name = 'Rory';
30 kitty.numLives--;
```

Interfaces

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

```
1 // first interface
2 // parameter has to be an object and has to have a label property
3 function printLabel(labelledObj: {label: string}) {
4   console.log(labelledObj.label);
5 }
6
7 let myObj = {size: 10, label: 'Size 10 Object'};
8 printLabel(myObj);
9
10 // with interface keyword
```

```
11 interface LabelledValue {
12     label: string;
13 }
14
15 function printLabel(labelledObj: LabelledValue) {
16     console.log(labelledObj.label);
17 }
18
19 // optional properties
20 interface SquareConfig {
21     color?: string;
22     width?: number;
23 }
24
25 function createSquare(config: SquareConfig): {color: string; area: number} {
26     let newSquare = {color: 'white', area: 100};
27     if (config.color) {
28         newSquare.color = config.color;
29     }
30     if (config.width) {
31         newSquare.area = config.width * config.width;
32     }
33     return newSquare;
34 }
35
36 let mySquare = createSquare({color: 'black'});
37
38 // additional properties
39 interface SquareConfig {
40     color?: string;
41     width?: number;
42     [propName: string]: any;
43 }
44
45 // function types
46 interface SearchFunc {
47     (source: string, subString: string): boolean;
48 }
49
50 let mySearch: SearchFunc;
51 mySearch = function(source: string, subString: string) {
52     return source.search(subString) !== -1;
53 }
54
55 // indexable types
56 interface StringArray {
57     [index: number]: string;
58 }
59
```



```
60 let myArray: StringArray;
61 myArray = ['Bob', 'Fred'];
62 let myStr: string = myArray[0];
63
64 // class types
65 interface ClockInterface {
66     currentTime: Date;
67     setTime(d: Date);
68 }
69
70 interface ClockConstructor {
71     new (hour: number, minute: number);
72 }
73
74 class Clock implements ClockConstructor, ClockInterface {
75     currentTime: Date;
76     setTime(d: Date) {
77         this.currentTime = d;
78     }
79     constructor(h: number, m: number) {}
80 }
81
82 // extending interfaces
83 interface Shape {
84     color: string;
85 }
86
87 interface PenStroke {
88     penWidth: number;
89 }
90
91 interface Square extends Shape, PenStroke {
92     sideLength: number;
93 }
94
95 let square = <Square>{};
96 square.color = 'blue';
97 square.sideLength = 10;
98 square.penWidth = 5.0;
99
100 // hybrid types
101 interface Counter {
102     (start: number): string;
103     interval: number;
104     reset(): void;
105 }
106
107 function getCounter(): Counter {
108     let counter = <Counter>function (start: number) { };
```

```

109     counter.interval = 123;
110     counter.reset = function () { };
111     return counter;
112 }
113
114 let c = getCounter();
115 c(10);
116 c.reset();
117 c.interval = 5.0;
118
119 // interfaces extending classes
120 class Control {
121     private state: any
122 }
123
124 interface SelectableControl extends Control {
125     select(): void;
126 }
127
128 class Button extends Control {
129     select() { }
130 }

```

Classes

```

1 // classes
2 class Greeter {
3     greeting: string;
4     constructor(message: string) {
5         this.greeting = message;
6     }
7     greet() {
8         return 'Hello, ' + this.greeting;
9     }
10 }
11
12 let greeter = new Greeter('world');
13
14 // inheritance
15 class Animal {
16     name: string;
17     constructor(theName: string) { this.name = theName; }
18     move(distanceInMeters: number = 0) {
19         console.log(`${this.name} moved ${distanceInMeters}m.`);
20     }
21 }
22
23 class Snake extends Animal {
24     constructor(name: string) { super(name); }

```

```

25     move(distanceInMeters = 5) {
26         console.log('Slithering...');
27         super.move(distanceInMeters);
28     }
29 }
30
31 class Horse extends Animal {
32     constructor(name: string) { super(name); }
33     move(distanceInMeters = 45) {
34         console.log('Galloping...');
35         super.move(distanceInMeters);
36     }
37 }
38
39 let sam = new Snake('Sammy the Python');
40 let tom: Animal = new Horse('Tommy the Palomino');
41
42 sam.move(); // Slithering... Sammy the Python moved 5m.
43 tom.move(34); // Galloping... Tommy the Palomino moved 34m.
44
45 // public, private, and protected modifiers
46 // public by default
47
48 // private
49 class Animal {
50     private name: string;
51     constructor(theName: string) { this.name = theName; }
52 }
53
54 new Animal('Cat').name; // error
55
56 // protected
57 class Person {
58     protected name: string;
59     constructor(name: string) { this.name = name; }
60 }
61
62 class Employee extends Person {
63     private department: string;
64     constructor(name: string, department: string) {
65         super(name);
66         this.department = department;
67     }
68     public getElevatorPitch() {
69         return `Hello, my name is ${this.name} and I work in
70         ${this.department}.`;
71     }
72 }

```

```

73 let howard = new Employee('Howard', 'Sales');
74 console.log(howard.getElevatorPitch()); // Hello, my name is Howard and I
    work in Sales.
75 console.log(howard.name); // error
76
77 // parameter properties
78 class Animal {
79     // shorthand to create and initialize the "name" member
80     constructor(private name: string) { }
81 }
82
83 // static properties
84 // visible on the class itself rather than on the instances
85 class Grid {
86     static origin = {x: 0, y: 0};
87     calculateDistanceFromOrigin(point: {x: number, y: number}) {
88         let xDist = (point.x - Grid.origin.x);
89         let yDist = (point.y - Grid.origin.y);
90         return Math.sqrt(xDist * xDist + yDist * yDist) / this.scale;
91     }
92     constructor (public scale: number) { }
93 }
94
95 let grid1 = new Grid(1.0); // 1x scale
96 let grid2 = new Grid(5.0); // 5x scale
97
98 console.log(grid1.calculateDistanceFromOrigin({x: 10, y: 10}));
99 console.log(grid2.calculateDistanceFromOrigin({x: 10, y: 10}));
100
101 // abstract classes
102 abstract class Animal {
103     // must be implemented in the derived classes
104     abstract makeSound(): void;
105     move(): void {
106         console.log('roaming the earth...');
107     }
108 }

```

Functions

```

1 // writing the function type
2 let myAdd: (x: number, y: number) => number =
3     function(x: number, y: number): number { return x + y; }
4
5 // optional parameters
6 function buildName(firstName: string, lastName?: string) {
7     return lastName? firstName + ' ' + lastName : lastName;
8 }
9

```

```

10 // default parameters
11 function buildName(firstName: string, lastName = 'Smith') {
12     return firstName + ' ' + lastName;
13 }
14
15 // rest parameters
16 function buildName(firstName: string, ...restOfName: string[]) {
17     return firstName + ' ' + restOfName.join(' ');
18 }
19
20 // lambdas and using "this"
21 let deck = {
22     suits: ['hearts', 'spades', 'clubs', 'diamonds'],
23     cards: Array(52),
24     createCardPicker: function() {
25         return function() {
26             let pickedCard = Math.floor(Math.random() * 52);
27             let pickedSuit = Math.floor(pickedCard / 13);
28             // "this" does reference "window" instead of "deck"
29             return {
30                 suit: this.suits[pickedSuit],
31                 card: pickedCard % 13
32             };
33         };
34     }
35 };
36
37 let cardPicker = deck.createCardPicker();
38 let pickedCard = cardPicker();
39
40 alert('card: ' + pickedCard.card + ' of ' + pickedCard.suit);
41
42 let deck = {
43     suits: ['hearts', 'spades', 'clubs', 'diamonds'],
44     cards: Array(52),
45     createCardPicker: function() {
46         // notice: the line below is now a lambda, allowing us to capture "this"
47         // earlier
48         return () => {
49             let pickedCard = Math.floor(Math.random() * 52);
50             let pickedSuit = Math.floor(pickedCard / 13);
51             return {
52                 suit: this.suits[pickedSuit],
53                 card: pickedCard % 13
54             };
55         };
56     };
57 }

```

```

58 // overloads
59 let suits = ['hearts', 'spades', 'clubs', 'diamonds'];
60
61 function pickCard(x: {suit: string; card: number; }[]): number;
62 function pickCard(x: number): {suit: string; card: number; };
63 function pickCard(x): any {
64     if (typeof x == 'object') {
65         let pickedCard = Math.floor(Math.random() * x.length);
66         return pickedCard;
67     } else if (typeof x == 'number') {
68         let pickedSuit = Math.floor(x / 13);
69         return {suit: suits[pickedSuit], card: x % 13};
70     }
71 }
72
73 let myDeck = [
74     {suit: 'diamonds', card: 2},
75     {suit: 'spades', card: 10},
76     {suit: 'hearts', card: 4}
77 ];
78 let pickedCard1 = myDeck[pickCard(myDeck)];
79 let pickedCard2 = pickCard(15);

```

Generics

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

```

1 // we loose type information hier
2 function identity(arg: any): any {
3     return arg;
4 }
5
6 // use type variable
7 function identity<T>(arg: T): T {
8     return arg;
9 }
10
11 // function declaration
12 let myIdentity: <T>(arg: T) => T = identity;
13
14 let output = identity<string>('myString'); // type of output will be "string"
15 // with type argument inference compiler will set value of "T"
16 let output = identity('myString'); // type of output will be "string"
17
18 // generic interface
19 interface GenericIdentityFn {
20     <T>(arg: T): T;
21 }
22

```

```

23 function identity<T>(arg: T): T {
24     return arg;
25 }
26
27 let myIdentity: GenericIdentityFn = identity;
28
29 // with type information
30 interface GenericIdentityFn<T> {
31     (arg: T): T;
32 }
33
34 function identity<T>(arg: T): T {
35     return arg;
36 }
37
38 let myIdentity: GenericIdentityFn<number> = identity;
39
40 // generic classes
41 class GenericNumber<T> {
42     zeroValue: T;
43     add: (x: T, y: T) => T;
44 }
45
46 let myGenericNumber = new GenericNumber<number>();
47 myGenericNumber.zeroValue = 0;
48 myGenericNumber.add = function(x, y) { return x + y; };
49
50 // generic constraints
51 interface Lengthwise {
52     length: number;
53 }
54
55 // constraint is, that T has "length" member
56 function loggingIdentity<T extends Lengthwise>(arg: T): T {
57     console.log(arg.length);
58     return arg;
59 }
60
61 // using type parameters in generic constraints
62 function copyFields<T extends U, U>(target: T, source: U): T {
63     for (let id in source) {
64         target[id] = source[id];
65     }
66     return target;
67 }
68
69 let x = {a: 1, b: 2, c: 3, d: 4};
70
71 copyFields(x, {b: 10, d: 20}); // ok

```

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

Enum

```
1 | enum Direction {Up = 1, Down, Left, Right}
2 |
3 | // reverse mapping
4 | enum Enum {A}
5 | let a = Enum.A; // 0
6 | let nameOfA = Enum[Enum.A]; // "A"
7 |
8 | // const enum
9 | const enum Directions {Up, Down, Left, Right}
10 | // generated code (without possibility to lookup names):
11 | var directions = [0 /* Up */, 1 /* Down */, 2 /* Left */, 3 /* Right */];
```

Type Inference

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

Type Compatibility

```
1 | interface Named {
2 |     name: string;
3 | }
4 |
5 | class Person {
6 |     name: string;
7 | }
8 |
9 | let p: Named;
10 | p = new Person(); // ok, because of structural typing
11 |
12 | let x: Named;
13 | // y's inferred type is {name: string, location: string}
14 | let y: {name: 'Alice', location: 'Seattle'};
15 | x = y;
16 |
17 | let x = (a: number) => 0;
18 | let y = (b: number, s: string) => 0;
19 |
20 | y = x; // ok
21 | x = y; // error
```

Symbols


```

1  let sym1 = Symbol();
2  // optional string key
3  let sym2 = Symbol('key');
4
5  // symbols are immutable and unique
6  let sym3 = Symbol('key');
7  sym2 === sym3; // false
8
9  // can be used as keys for object properties
10 let obj = {
11   [sym]: 'value'
12 };
13 console.log(obj[sym]); // value

```

Iterators and Generators

```

1  // iterables
2  let someArray = [1, 'string', false];
3  for (let i in someArray) {
4   console.log(i); // 0, 1, 2
5  }
6  for (let i of someArray) {
7   console.log(i); // 1, "string", false
8  }

```

Namespaces

```

1  namespace Validation {
2   export interface StringValidator {
3     isAcceptable(s: string): boolean;
4   }
5
6   const lettersRegexp = /^[A-Za-z]+$/;
7   const numberRegexp = /^[0-9]+$/;
8
9   export class LettersOnlyValidator implements StringValidator {
10    isAcceptable(s: string) {
11     return lettersRegexp.test(s);
12    }
13   }
14
15   export class ZipCodeValidator implements StringValidator {
16    isAcceptable(s: string) {
17     return numberRegexp.test(s);
18    }
19   }
20 }
21

```

```

22 let strings = ['Hello', '98052', '101'];
23
24 let validators: {[s: string]: Validation.StringValidator; } = {};
25 validators['ZIP code'] = new Validation.ZipCodeValidator();
26 validators['Letters only'] = new Validation.LettersOnlyValidator();
27
28 for (let s of strings) {
29     for (let name in validators) {
30         console.log(`"${s}" - ${validators[name].isAccetable(s) ? 'matches' :
31             'does not match'} ${name}`);
32     }
33 }
34 // splitting across files
35 // validation.ts
36 namespace Validation {
37     // ...
38 }
39
40 // letters-only-validator.ts
41 /// <reference path="validation.ts" />
42 namespace Validation {
43     // ...
44 }
45
46 // aliases
47 namespace Shapes {
48     export namespace Polygons {
49         export class Triangle { }
50         export class Square { }
51     }
52 }
53
54 import polygons = Shapes.Polygons;
55 let sq = new polygons.Square();
56
57 // ambient namespaces
58 declare namespace D3 {
59     export interface Selectors {
60         select: {
61             (selector: string): Selection;
62             (element: EventTarget): Selection;
63         }
64     }
65
66     export interface Event {
67         x: number;
68         y: number;
69     }

```

```

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;

```

```

17     }
18 }
19
20 class SmartObject implements Disposable, Activatable {
21     constructor() {
22         setInterval(() => console.log(this.isActive + ' : ' + this.isDisposed),
23             500);
24     }
25
26     interact() {
27         this.activate();
28     }
29
30     isDisposed: boolean = false;
31     dispose: () => void;
32
33     isActive: boolean = false;
34     activate: () => void;
35     deactivate: () => void;
36 }
37
38 function applyMixins(derivedCtor: any, baseCtors: any[]) {
39     baseCtors.forEach(baseCtor => {
40         Object.getOwnPropertyNames(baseCtor.prototype).forEach(name => {
41             derivedCtor.prototype[name] = baseCtor.prototype[name];
42         });
43     });
44 }
45
46 applyMixins(SmartObject, [Disposable, Activatable]);
47
48 let smartObj = new SmartObject();
49 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 browser
- 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

```
1 export const counter: Reducer<number> = (state: number = 0, action: Action) =>
  {
2   switch (action.type) {
3     case 'INCREMENT':
4       return state + 1;
5     case 'DECREMENT':
6       return state - 1;
7     default:
8       return state;
9   }
10  };
```

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 underlying 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 underlying 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 functions 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 functions: Pure reducers specify how the state tree is transformed by actions