Building A JavaScript Framework

Table of Contents

Table of Contents

Table of Contents

Introduction

This book is a guide to building a JavaScript framework. It'll teach you how to build a framework and draw on real-world code from projects like jQuery

Along the way we'll explore some fundamental parts of modern JavaScript:

- Browser capability detection
- Clean, reusable API design
- Benchmarking and performance
- Writing minifier-friendly JavaScript
- Using GitHub!

The framework we'll build is called Turing and is available on GitHub: turing.js.

Framework Style

If you embark on an involved open source or private project, you're likely to work with other people. It's important to be upfront about the goals of the project and the style of development.

These are the practices I will use to develop this framework:

- **Verbose**: Variable and method names should be verbose so things are easy to find and understand
- **Portable**: Browsers and console should be catered for
- **Explicit**: Code should be quick to understand
- **Comments**: Let's keep comment noise down. Comments should be succinct. TODO and FIXME are acceptable.
- **Simple**: Keep code simple. Let's not bore readers!
- **Indentation**: Two spaces
- **Semicolons**: People might want to minify this library — let's keep simicolons!
- **Quality**: JsLint and reader comments!
- **Testing**: Test first development for both browsers and console
- **Versioning**: GitHub to the rescue

High Level Framework Structure

The first question to ask about a JavaScript framework's structure is: how self-contained is it? In 2005 we were blown away by Ajax and the yellow fade technique, so people flocked to libraries that made those techniques easy. Now in 2010 we're writing server-side JavaScript and creating sophisticated front-end behaviour. We can no-longer afford to use frameworks that aren't careful about their namespacing.

Take a look at the current stable prototype.js. It modifies the prototypes of a lot of native objects. It also provides a lot of top-level objects. The BBC specifically designed Glow to avoid this, and literally everything is namespaced. This feels strange if you're used to Prototype, because Prototype attempts to simplify browser-based JavaScript. Prototype makes complex Array manipulation much easier cross-browser, but with Glow you need to remember to use `glow.lang.toArray` and other utility methods.

The lesson here is that you trade off usability to play nice with other frameworks. Due to the way JavaScript works though, it's possible to use both approaches — our library could have configuration options to extend native objects.

This framework will be more like Glow — this will remove a lot of hidden magic when using it. People using it to learn JavaScript should be able to see the difference between what browsers and CommonJS provide.

Another interesting point about Prototype is it quickly defines high-level structural code which it reuses internally. It defines `Object.extend` and `Class`, then reuses these to build fundamental features:

```
var  Hash  = Class .create  (Enumerable  , (function  ()  {
   function   initialize  (object ) {
     this ._object   = Object  .isHash  (object )  ?  object  .toObject  ()  :  Object  .clone  (object );
   }
```

Helper Methods

MooTools, jQuery and Prototype all define helpers to reduce the effort required to call commonly used functions:

```
// Prototype
function   $H (object )  {
   return   new  Hash (object );
};

// MooTools
function   $H (object ){
   return   new  Hash (object );
};
```

It would be nice to include quick access to helper methods, but as I said previously where `turing.js` begins and ends needs to be clear to the newcomer. Therefore, if these are to be used they should be succinct but clear.

If you taught someone JavaScript with jQuery, would they even realise browsers don't have `$()` ?

Initialisation

Most frameworks have wrappers for initialisation and metadata about the library. MooTools and Prototype use one broadly similar approach, then jQuery and Glow use another.

```
var  MooTools   = {
   'version'  :  '1.2.5dev'  ,
   'build'  :  '%build%'
};

var  Prototype   = {
   Version  :  '<%= PROTOTYPE_VERSION %>'      ,
   ...
}

(function   (  window  ,  undefined   )  {
   var  jQuery   =  function   (  selector  ,  context   )  {
         // The jQuery object is actually just the init constructor 'enhanced'
         return   new  jQuery  .fn .init (  selector  ,  context   );
      },
      ...
      jquery  :  "@VERSION"  ,
      ...
   }

   // Expose jQuery to the global object
   window  .jQuery   =  window  .$  =  jQuery  ;
})( window );
```

Glow and jQuery both use an anonymous function, then expose themselves by writing an attribute to `window`. This is the approach I'll use for turing.js.

Modules and Plugins

jQuery, MooTools and Glow have tried hard to be modular. Let's use a similar approach, with a file naming scheme like this:

- turing.core.js
- turing.functional.js

After creating a `turing` variable that will be exposed to the global scope, we can define our modules on it as functions or objects.

Let's Get Coding

I'm going to use my riot.js library to write unit tests, because it's a simple unit testing library that is pure JavaScript.

You might think that testing a library framework stub is pointless, but we still need to make sure it sets things up properly in browsers and in the console. I'm going to run my tests in Rhino and Firefox.

The core test should check the following:

- `turing` is instantiated
- `turing` has properties we can read — let's set a version number

The test code looks like this:

```
Riot.context('turing.core.js', function() {
  given('the turing object', function() {
    should('be global and accessible', turing).isNotNull();
    should('return a VERSION', turing.VERSION).isNotNull();
    should('be turing complete', true).isTrue();
  });
});
```

Putting this together, we get:

```
(function(global) {
  var turing = {
    VERSION: '0.0.1',
    lesson: 'Part 1: Library Architecture'
  };

  if (global.turing) {
    throw new Error('turing has already been defined');
  } else {
    global.turing = turing;
  }
})(typeof window === 'undefined' ? this : window);
```

Here's how it looks in Rhino:

```
js> load('turing.core.js');
js> print(turing.VERSION);
```

```
0.0 .1
js > print (turing .lesson );
Part  1: Library  Architecture
```

And in a browser:

```
>>>  turing
Object  {  VERSION  ="0.0.1"  ,  more ...}
```

Object Oriented JavaScript

Introduction

Not all JavaScript frameworks provide classes. Douglas Crockford discusses the *classical object model* in Classical Inheritance in JavaScript. It's an excellent discussion of ways to implement inheritance in JavaScript. Later, he wrote Prototypal Inheritance in JavaScript in which he basically concludes prototypal inheritance is a strong enough approach without the classical object model.

So why do JavaScript libraries provide tools for OO programming? The reasons vary depending on the author. Some people like to ape an object model from their favourite language. Prototype is heavily Ruby inspired, and provides Class which can be useful for organising your own code. In fact, Prototype uses `Class` internally.

In this chapter I'm going to explain prototypal inheritance and OO, and start to create a class for OO in JavaScript. This will be used by our framework, turing.js.

Objects and Classes vs. Prototype Classes

Objects are… everything, so some languages attempt to treat everything as an object. That means a number is an object, a string is an object, a class definition is an object, an instantiated class is an object. The distinction between classes an objects is interesting — these languages treat classes as objects, and use a more basic object model to implement classes. Remember: it's *object oriented programming* not *class oriented*.

So does that mean JavaScript really needs classical classes? If you're a Java or Ruby programmer you might be surprised to find JavaScript doesn't have a `class` keyword. That's OK though! We can build our own features if we need them.

Prototype Classes

Prototype classes look like this:

```
function Vector(x, y) {
  this.x = x;
  this.y = y;
}

Vector.prototype.toString = function() {
  return 'x: ' + this.x + ', y: ' + this.y;
}

v = new Vector(1, 2);
// x: 1, y: 2
```

If you're not used to JavaScript's object model, the first few lines might look strange. I've defined a function called `Vector`, then said `new Vector()`. The reason this works is that `new` creates a new object and then runs the function `Vector`, with `this` set to the new object.

The `prototype` property is where you define instance methods. This approach means that if you instantiate a vector, then add new methods to the `prototype` property, the old vectors will get the new methods. Isn't that amazing?

```
Vector.prototype.add = function(vector) {
  this.x += vector.x;
  this .y  +=  vector .y;
  return   this ;
}

v.add (new  Vector (5,  5));
// x: 6, y: 7
```

Prototypal Inheritance

There's no formal way of implementing inheritance in JavaScript. If we wanted to make a `Point` class by inheriting from `Vector`, it could look like this:

```
function   Point (x,  y,  colour )  {
  Vector  .apply (this ,  arguments  );
  this .colour   =  colour ;
}

Point .prototype   = new  Vector ;
Point .prototype  .constructor   =  Point ;

p  =  new  Point (1,  2,  'red' );
p.colour ;
// red
p.x;
// 1
```

By using `apply`, `Point` can call `Vector`'s constructor. You might be wondering where `prototype.constructor` comes from. This is a property that allows you to specify the function that creates the object's prototype.

When creating your own objects, you also get some methods for free that descend from `Object`. Examples of these include `toString` and `hasOwnProperty`:

```
p.hasOwnProperty   ('colour'  );
// true
```

Prototypal vs. Classical

There are multiple patterns for handling prototypal inheritance. For this reason it's useful to abstract it, and offer extra features beyond what JavaScript has as standard. Defining an API for classes keeps code simpler and makes it easer for people to navigate your code.

The fact that JavaScript's object model splits up portions of a class can be visually noisy. It might be attractive to wrap entire classes up in a definite start and end. Since this is a *teaching framework*, wrapping up classes in discrete and readable chunks might be beneficial.

A Class Model Implementation Design

The previous example in Prototype looks like this:

```
Vector   = Class .create  ({
  initialize  :  function  (x,  y)  {
    this .x  =  x;
    this .y  =  y;
  },

  toString  :  function  ()  {
    return   'x: '   +  this .x  +  ', y: '   +  this .y;
  }
});

Point  = Class .create (Vector , {
  initialize  : function  ($super , x, y, colour ) {
    $super (x, y);
    this .colour  = colour ;
  }
});
```

Let’s create a simplified version of this that we can extend in the future. We’ll need the following:

1. The ability to extend classes with new methods by copying them
2. Class creation: use of `apply` and `prototype.constructor` to run the constructors
3. The ability to determine if a parent class is being passed for inheritance
4. Mixins

Extend

You’ll find `extend` littered through Prototype. All it does is copies methods from one `prototype` to another. This is a good way to really see how prototypes can be manipulated — it’s as simple as you think it is.

The essence of `extend` is this:

```
for  (var  property   in  source )
  destination  [property ] = source [property ];
```

Class Creation

A `create` method will be used to create new classes. It will need to handle some setup to make inheritance possible, much like the examples above.

```
// This would be defined in our "oo" namespace
create : function  (methods ) {
  var  klass  = function  ()  { this .initialize  .apply (this ,  arguments  );  };

  // Copy the passed in methods
  extend (klass .prototype  , methods );

  // Set the constructor
  klass .prototype  .constructor   = klass ;

  // If there's no initialize method, set an empty one
  if  (!klass .prototype  .initialize  )
    klass .prototype  .initialize   = function  (){};

  return  klass ;
}
```

Classes in More Depth

The `initialize` method is our way of saying *call this method when you set up my class*.

Turing’s `Class.create` method sets up classes. During the setup it defines a function that will be called when the class is instantiated. So when you say `new`, it will run `initialize`. The code behind this is very simple:

```
 create : function () {
     var methods = null ,
         parent = undefined ,
         klass = function () {
           this .initialize .apply (this , arguments );
         };
```

I sometimes feel like `apply` is magical, but it's not really *magic* in the negative programmer sense of the word — it doesn't hide too much from you. In this case it just calls your `initialize` method against the newly created class using the supplied arguments. Again, the `arguments` variable seems like magic… but that's just a helpful variable JavaScript makes available when a function is called.

Because I've made this contract — that all objects will have initialize methods — we need to define one in cases where classes don't specify initialize:

```
    if (!klass .prototype .initialize )
       klass .prototype .initialize = function (){};
```

Syntax Sugar * Extend === Mixin

It would be cool to be able to mix other object prototypes into our class. Ruby does this and I've often found it useful. The syntax could look like this:

```
var MixinUser = turing .Class ({
  include : User ,

  initialize : function (log ) {
    this .log = log ;
  }
});
```

Mixins should have some simple rules to ensure the resulting object compositions are sane:

- Methods should be included from the specified classes
- The `initialize` method should not be overwritten
- Multiple includes should be possible

Since our classes are being run through `turing.oo.create`, we can easily look for an `include` property and include methods as required. Rather than including the bulk of this code in `create`, it should be in another `mixin` method in `turing.oo` to keep `create` readable.

To satisfy the rules above (which I've turned into unit tests), this pseudo-code is required:

```
 mixin : function (klass , things ) {
     if "there are some valid things" {
       if "the things are a class" {
         "use turing.oo.extend to copy the methods over"
       } else if "the things are an array" {
         for "each class in the array" {
           "use turing.oo.extend to copy the methods over"
         }
       }
     }
   },
```

Super

We need to be able to inherit from a class and call methods that we want to override. Given a `User` class (from the fixtures/example_classes.js file), we can inherit from it to make a `SuperUser`:

```
var  User  = turing .Class ({
  initialize  : function  (name ,  age ) {
    this .name  = name ;
    this .age   = age ;
  },

  login : function  ()  {
    return  true ;
  },

  toString  : function  ()  {
    return  "name: "   + this .name  + ", age: "   + this .age ;
  }
});

var  SuperUser   = turing .Class (User ,  {
  initialize  : function  ()  {
    // Somehow call the parent's initialize
  }
});
```

A test to make sure the parent's `initialize` gets called is simple enough:

```
 given ('an inherited class that uses super'       , function  ()  {
    var  superUser   = new  SuperUser  ('alex' ,  104 );
    should  ('have run super()'    ,  superUser  .age ). equals  (104 );
  });
```

If I run this without a `super` implementation, I get this:

```
Given an inherited class that uses super
  - should have run super(): 104 does not equal: undefined
```

To fix this all we need to do is call a previously defined method. Hey, it's time for `apply` again!

```
var  SuperUser   = turing .Class (User ,  {
  initialize  : function  ()  {
    User .prototype  .initialize  .apply (this ,  arguments  );
  }
});
```

This isn't perfect though. Most languages make their `super` implementation simpler for the caller — forcing people to use `apply` like this is unwieldy. One way around this is to make the parent class prototype available and add a super method to the class. The super method can simply use `apply` in the same manner. The only downside is you have to specify the method name:

```
var  SuperUser   = turing .Class (User ,  {
  initialize  : function  ()  {
    this .$super  ('initialize'   ,  arguments  );
  },

  toString  : function  ()  {
    return  "SuperUser: "    + this .$super  ('toString'  );
  }
});
```

This is simple, lightweight, and easy to understand. The method name could be inferred by other means, but this would complicate the library beyond the scope of this book (meaning if you can do it and make it cross-browser then cool!)

Conclusion

Now we’ve got a simple, readable OO class. This will allow us to structure other parts of Turing in a reusable way. I hope the last two examples have demonstrated that JavaScript’s simplicity allows you to define your own behaviour for things that feel like fundamental language features.

Functional Programming

The name *functional programming* is annoying because it reminds novices of *procedural programming*, which people learn first then discard in favour of object oriented programming. Forget all that for a bit.

Functional programming is about:

- Describing problems rather than focusing on the mechanics of their solution
- Treating functions as first class citizens, and manipulating them like variables
- Avoiding state and mutable data

There are functional programming languages like Erlang and Haskell. JavaScript isn't strictly functional, but neither are Ruby, Python, or Java. However, these languages use ideas from functional programming to simplify common programming tasks.

Functional languages usually focus on lists, and treat functions as *first class* and have useful features like closures. JavaScript is actually pretty good at functions and closures, and JavaScript arrays and objects are similar to lists and property lists in lisp-like languages.

I once saw a talk by Tom Stuart called Thinking Functionally in Ruby. It has some animations that *really* make functional programming paradigms easy to understand. If you're still finding the concept nebulous try watching his presentation.

Iterators

JavaScript frameworks use `each` a lot. Some frameworks define classes early on, but almost all define this basic tool for iteration.

Ruby programmers use `each` all the time:

```
[1, 2, 3].each { |number| puts number }
# 1
# 2
# 3
```

This sends a block to `each`, and `each` runs the block multiple times. Enumerable uses `each` to create lots of other methods that are inspired by functional languages. Any collection-style object can mixin Enumerable to get all those methods for free.

Equivalent JavaScript could look like this:

```
Array.prototype.each = function(callback) {
  for (var i = 0; i < this.length; i++) {
    callback(this[i]);
  }
}

[1, 2, 3].each(function(number) {
  print(number);
});
```

However, JavaScript actually has `Array.forEach`, `Array.prototype.forEach`, `for (var i in objectWithIterator)` and even more ways to iterate. So why do frameworks bother defining their own method? One of the reasons you see jQuery.each and each in Prototype is because browser support is inconsistent.

You can see the source for jQuery's each in core.js.

Prototype's implementation uses `forEach` if it exists:

```
(function () {
  var arrayProto  = Array .prototype ,
      slice  = arrayProto  .slice ,
      _each  = arrayProto  .forEach ; // use native browser JS 1.6 implementation if available
```

Underscore uses a similar approach:

```
 // The cornerstone, an each implementation.
  // Handles objects implementing forEach, arrays, and raw objects.
  // Delegates to JavaScript 1.6's native forEach if available.
  var each  = _.forEach  = function (obj , iterator , context ) {
    try {
      if (nativeForEach   && obj .forEach  === nativeForEach  ) {
        obj .forEach (iterator , context );
      } else if (_.isNumber (obj .length )) {
        for (var i = 0, l = obj .length ; i < l; i++) iterator .call (context , obj [i], i, obj );
      } else {
        for (var key in obj ) {
          if (hasOwnProperty  .call (obj , key )) iterator .call (context , obj [key ], key , obj );
        }
      }
    } catch (e) {
      if (e != breaker ) throw e;
    }
    return obj ;
  };
```

This approach uses JavaScript's available datatypes and features, rather than mixing an Enumerable-style class into objects and `Array`.

Benchmarks

I've written some benchmarks to test each implementation. You can view them here: test.html, and iteratortest.js.

Each will form a cornerstone of Turing's functional programming features, so let's create a benchmark to see if the native function is really faster.

	Rhino	Node	Firefox	Safari	Chrome	Opera	IE8	IE7	IE6
eachNative	**1428ms**	69ms	**709ms**	114ms	62ms	1116ms			
eachNumerical	2129ms	**55ms**	904ms	**74ms**	**58ms**	**1026ms**	**3674ms**	**10764ms**	**6840ms**
eachForIn	4223ms	309ms	1446ms	388ms	356ms	2378ms	4844ms	21782ms	14224ms

The native method performs well, and generally close to the simple for loop. This probably explains why most JavaScript library implementors use it when it's there. And `for ... in` performs so terribly in Internet Explorer that we need to be careful about using it.

API Design

An important consideration is the API design for functional features. The Prototype library modifies `Object` and `Array`'s prototypes. This makes the library easy to use, but makes it difficult to use concurrently with other libraries: it isn't safely namespacing.

Underscore has clearly namespaced design, with optional use for what it calls functional or object-oriented (which allows chained calls).

Our library could look like this:

```
turing.enumerable.each([1, 2, 3], function(number) { number + 1; });
turing.enumerable.map([1, 2, 3], function(number) { return number + 1; });
// 2, 3, 4
```

These methods could be mapped to shorthands later on.

Tests

A basic test of `each` and `map` should ensure arrays are iterated over:

```
Riot.context('turing.enumerable.js', function() {
  given('an array', function() {
    var a = [1, 2, 3, 4, 5];
    should('iterate with each', function() {
      var count = 0;
      turing.enumerable.each(a, function(n) { count += 1; });
      return count;
    }).equals(5);

    should('iterate with map', function() {
      return turing.enumerable.map(a, function(n) { return n + 1; });
    }).equals([2, 3, 4, 5, 6]);
  });
});
```

Objects should also work:

```
given('an object', function() {
  var obj = { one: '1', two: '2', three: '3' };
  should('iterate with each', function() {
    var count = 0;
    turing.enumerable.each(obj, function(n) { count += 1; });
    return count;
  }).equals(3);

  should('iterate with map', function() {
    return turing.enumerable.map(obj, function(n) { return n + 1; });
  }).equals(['11', '21', '31']);
});
```

The resulting implementation is based heavily on Underscore, because it's easy to understand and my benchmarks show it's pretty smart. View the code here: turing.enumerable.js

Functional Methods

Let's build more functional methods that build on `each`. I'll draw on inspiration from Underscore and Prototype, not to mention JavaScript's more recent `Array.prototype` methods.

Filter

Filter allows you to remove values from a list:

```
turing.enumerable.filter([1, 2, 3, 4, 5, 6], function(n) { return n % 2 == 0; });
```

```
// 2,4,6
```

That means the implementation needs to:

1. Check if there’s a native `filter` method and use it if possible
2. Else use `turing.enumerable.each`
3. Filter objects into multi-dimensional arrays if required

The tests need to check that both arrays and objects are handled. We’ve already seen this approach:

```
Riot .context ('turing.enumerable.js'    , function () {
  given ('an array'  , function () {
    var a = [1, 2, 3, 4, 5];

    should ('filter arrays'   , function () {
      return  turing .enumerable  .filter (a, function (n) { return  n % 2 == 0; });
    }). equals ([2, 4]);
  });

  given ('an object'  , function () {
    var obj = { one : '1' , two : '2' , three : '3' };

    should ('filter objects and return a multi-dimensional array'          , function () {
      return  turing .enumerable  .filter (obj , function (v, i) { return  v < 2; })[0][0];
    }). equals ('one' );
  });
});
```

I’ve tried to be sensible about handling both objects and arrays. Underscore supports filtering objects, but returns a slightly different result (it just returns the value instead of key/value).

Detect

Detect is slightly different to `filter` because there isn’t an ECMAScript method. It’s easy to use though:

```
turing .enumerable  .detect (['bob' , 'sam' , 'bill' ], function (name ) { return  name  ===  'bob' ; });
// bob
```

This class of methods is interesting because it requires an early break. You may have noticed that the `each` method had some exception handling that checked for `Break`:

```
each : function (enumerable  , callback , context ) {
  try {
    // The very soul of each
  } catch (e) {
    if (e != turing .enumerable  .Break ) throw  e;
  }

  return  enumerable  ;
}
```

Detect simply uses `each` with the user-supplied callback, until a truthy value is returned. Then it throws a `Break`.

Chaining

We need to be able to chain these calls if we can honestly say turing.js is useful. Chaining is natural when you’ve overridden `Array.prototype` like some libraries do, but seeing as we’re being good namespacers we need to create an API for it.

I'd like it to look like this (which is different to Underscore):

```
turing .enumerable  .chain ([1,  2,  3,  4]). filter (function  (n)  {  return   n  %  2  ==  0;  }). map (function  (n)
```

Chained functions are possible when each function returns an object that can be used to call the next one. If this looks confusing to you, it might help to break it down:

```
.chain ([1,  2,  3,  4])                               // Start a new "chain" using an array
.filter  (function  (n)  {  return   n  %  2  ==  0;  })   // Filter out odd numbers
.map (function  (n)  {  return   n  *  10 ;  })            // Multiply each number by 10
.values  ();                                           // Fetch the values
```

To make this possible we need a class with the following features:

- Store temporary values
- Runs appropriate methods from `turing.enumerable` by mapping the temporary value into the first argument
- After running the method, return `this` so the chain can continue

This is all easily possible using closures and `apply`:

```
// store temporary values in this.results
turing .enumerable  .Chainer   = turing .Class ({
  initialize  : function  (values )  {
    this .results   = values ;
  },

  values  : function  ()  {
    return   this .results  ;
  }
});

// Map selected methods by wrapping them in a closure that returns this each time
turing .enumerable  .each (['map' , 'detect'  , 'filter'  ], function  (methodName  )  {
  var  method   = turing .enumerable  [methodName  ];
  turing .enumerable  .Chainer .prototype  [methodName  ]  = function  ()  {
    var  args   = Array .prototype  .slice .call (arguments  );
    args .unshift  (this .results );
    this .results   = method .apply (this ,  args );
    return   this ;
  }
});
```

Conclusion

Now you know how to:

- Check for native methods that operate on collections of values
- Implement them using `each` where required
- Break early using an exception
- Chain methods using closures
- All in a safely namespaced API!

Selector and Selector Engines

Introduction

In this part I'll explain the basics behind selector engines, explore the various popular ones, and examine their characteristics. This is a major area for JavaScript web frameworks.

History

The importance and need for cross-browser selector APIs is a big thing. Whether the selectors are XPath or CSS, browser implementors are not consistent.

To understand why a selector engine is important, imagine working in the late 90s without a JavaScript framework:

```
document.all      // A proprietary property in IE4
document.getElementById('navigation');
```

The first thing that people realised was this could be shortened. Prototype does this with `$()`:

```
function $(element) {
  if (arguments.length > 1) {
    for (var i = 0, elements = [], length = arguments.length; i < length; i++)
      elements.push($(arguments[i]));
    return elements;
  }
  if (Object.isString(element))
    element = document.getElementById(element);
  return Element.extend(element);
}
```

Rather than simply aliasing `document.getElementById`, Prototype can search for multiple IDs and extends elements with its own features.

What we really wanted though was `getElementsBySelector`. We don't just think in terms of IDs or tag names, we want to apply operations to sets of elements. CSS selectors are how we usually do this for styling, so it makes sense that similarly styled objects should *behave* in a similar way.

Simon Willison wrote getElementsBySelector back in 2003. This implementation is partially responsible for a revolution of DOM traversal and manipulation.

Browsers provide more than just `getElementById` — there's also getElementsByClassName, getElementsByName, and lots of other DOM-related methods. Most non-IE browsers support searching using XPath expressions using evaluate.

Modern browsers also support querySelector and querySelectorAll. Again, these methods are limited by poor IE support.

Browser Support

Web developers consider browser support a necessary evil. In selector engines, however, browser support is a major concern. Easing the pain of working with the DOM and getting reliable results is fundamental.

I've seen some fascinating ways to work around browser bugs. Look at this code from Sizzle:

```
// Check to see if the browser returns elements by name when
// querying by getElementById (and provide a workaround)
(function (){
  // We're going to inject a fake input element with a specified name
  var form = document .createElement ("div" ),
  id = "script" + (new Date ()). getTime ();
  form .innerHTML = "<a name='" + id + "'/>" ;

  // Inject it into the root element, check its status, and remove it quickly
  var root = document .documentElement ;
  root .insertBefore ( form , root .firstChild );
```

It creates a fake element to probe browser behaviour. Supporting browsers is a black art beyond the patience of most well-meaning JavaScript hackers.

Performance

Plucking elements from arbitrary places in the DOM is useful. So useful that developers do it a lot, which means selector engines need to be fast.

Falling back to native methods is one way of achieving this. Sizzle looks for `querySelectorAll`, with code to support browser inconsistencies.

Another way is to use caching, which Sizzle and Prototype both do.

One way to measure selector performance is through a tool like slickspeed or a library like Woosh. These tools are useful, but you have to be careful when interpreting the results. Libraries like Prototype and MooTools extend elements, whereas pure selector engines like Sizzle don't. That means they might look slow compared to Sizzle but they're actually fully-fledged frameworks rather than a pure selector engine.

Other Selector Engines

I've been referencing popular frameworks and Sizzle so far, but there are other Sizzle-alikes out there. Last year there was a burst of several, perhaps in reaction to the first release of Sizzle. Sizzle has since been gaining popularity with framework implementors again, so I'm not sure if any really made an impact.

Peppy draws on inspiration from lots of libraries and includes some useful comments about dealing with caching. Sly has a lot of work on optimisation and allows you to expose the parsing of selectors. Both Sly and Peppy are fairly large chunks of code — anything comparable to Sizzle is not a trivial project.

API Design

Generally speaking, there are two kinds of APIs. One approach uses a function that returns elements that match a selector, but wraps them in a special class. This class can be used to chain calls that perform complex DOM searches or manipulations. This is how jQuery, Dojo and Glow work.

jQuery gives us `$()`, which can be used to query the document for nodes, then chain together calls on them.

Dojo has `dojo.query` which returns an Array-like dojo.NodeList. Glow is similar, with glow.dom.get

The second approach is where elements are returned and extended. Prototype and MooTools do this.

Prototype has `$()` for `getElementById` and `$$()` for querying using CSS or XPath selectors. Prototype extends elements with its own methods. MooTools behaves a lot like this. Both can work with strings or element references.

Turing has been designed much like Glow-style libraries, so we'll use this approach for our API. There's a lazy aspect to this design that I like — it might be possible to return unprocessed elements wrapped in an object, then only deal with masking browser inconsistencies when they're actually manipulated:

```
turing .dom .find ('.class' )                     // return elements wrapped in a class without looking at each of
.find ('a' )                                      // find elements that are links
.css ({  'background-color'    :  '#aabbcc'   })  // apply the style by actually processing elements
```

Goals

Because Turing exists as an educational framework, it's probably wise if we keep it simple:

- CSS selectors only (XPath could be a future upgrade or plugin)
- Limited pseudo-selectors. Implementing one or two would be nice just to establish the API for them
- Defer to native methods as quickly as possible
- Cache for performance
- Expose the parsing results like Sly does
- Reuse the wisdom of existing selector engines for patching browser nightmares

CSS Selectors

Before trying to parse anything, it's a good idea to become familiar with the input. CSS selectors aren't trivial, but to mitigate unnecessary complications I'll limit our library to a subset of CSS2 for now.

CSS2 selectors are explained in detail by in the spec: Selectors: Pattern Matching and Appendix G. Grammar of CSS 2.1. These documents are readable, don't be afraid to skim them!

I want to initially focus on the following syntaxes:

- `E` – Matches any element named `E`
- `E F` – Matches any element `F` that descends from `E`
- `.class` – Matches any element with the class `class`
- `E.class` – Matches any element named `E` with the class `class`
- `#id` – Matches any element with the id `id`
- `E#id` – Matches any element named `E` with the id `id`

Any one of these rules forms a *simple selector*. Simple selectors can be chained with *combinators*: white space, '>', and '+'.

Parsing and Searching Strategy

A good place to look for parsing strategies (other than existing selector engines) is browsers. The Mozilla developer site has an article entitled Writing Efficient CSS that actually explains how the style system matches rules.

It breaks up selectors into four categories:

1. ID Rules
2. Class Rules
3. Tag Rules
4. Universal Rules

The last part of a selector is called the *key selector*. The ancestors of the key selector are analysed to determine which elements match the entire selector. This allows the engine to ignore rules it doesn't need. The upshot of this is that `element#idName` would perform slower than `#idName`.

This algorithm isn't necessarily the fastest — many rules would rely on `getElementsByTagName` returning a lot of elements. However, it's an incredibly easy to understand and pragmatic approach.

Rather than branching off rule categories, we could just put the categories in an object:

```
 findMap   = {
     'id' :  function  (root ,  selector  )  {
     },

     'name and id'   :  function  (root ,  selector  )  {
     },

     'name'  :  function  (root ,  selector  )  {
     },

     'class'  :  function  (root ,  selector  )  {
     },

     'name and class'    :  function  (root ,  selector  )  {
     }
  };
```

Tokenizer

Tokens are just categorised strings of characters. This stage of a parser is called a *lexical analyser*. It sounds more complicated than it is. Given a selector, we need to:

- Normalise it to remove any extra white space
- Process it by some means to transform it into a sequence of instructions (tokens) for our parser
- Run the tokens through the parser against the DOM

We can model a token as a class like this:

```
function   Token (identity  ,  finder )  {
  this .identity   =  identity  ;
  this .finder     =  finder ;
}

Token .prototype  .toString   =  function  ()  {
  return   'identity: '    +  this .identity   +  ', finder: '    +  this .finder ;
};
```

The `finder` property is one of those keys in `findMap`. The `identity` is the original rule from the selector.

Scanner

Both Sly and Sizzle use a giant regular expression to break apart and process a selector. Sizzle calls this the `Chunker`.

Given the performance and flexibility of regular expressions in JavaScript, this is a good approach. However, I don't want to confuse readers with a giant regular expression. What we need is an intermediate approach.

Most programming languages have tools that generate tokenizers based on lexical patterns. Typically a lexical analyser outputs source code based on the output of a parser generator

Lexers were traditionally used for building programming language parsers, but we now live in a world filled with a gamut of computer-generated data. This means that you'll even find these tools cropping up in projects like nokogiri, a Ruby HTML and XML parser.

The power of lexers lies in the fact that there's a layer of abstraction between the programmer and the parser. Working on simple rules is easier than figuring out the finer implementation details.

Let's use an incredibly simplified version of a lexer to create the regular expression that drives the tokenizer. These rules can be based on the lexical scanner description in the CSS grammer specification.

It will be useful to embed regular expressions within other ones, and not worry about escaping them. Objects like this will drive the process:

```
macros  = {
  'nl' :          '\n|\r\n|\r|\f'  ,
  'nonascii'  :   '[^\0-\177]'  ,
  'unicode'  :    '\\[0-9A-Fa-f]{1,6}(\r\n|[\s\n\r\t\f])?'        ,
  'escape'  :     '#{unicode}|\\[^\n\r\f0-9A-Fa-f]'       ,
  'nmchar'  :     '[_A-Za-z0-9-]|#{nonascii}|#{escape}'        ,
  'nmstart'  :    '[_A-Za-z]|#{nonascii}|#{escape}'       ,
  'ident'  :      '[-@]?(#{nmstart})(#{nmchar})*'      ,
  'name'  :       '(#{nmchar})+'
};

rules  = {
  'id and name'   :     '(#{ident}##{ident})'    ,
  'id' :                '(##{ident})'   ,
  'class'  :            '(\\.#{ident})'   ,
  'name and class'   :  '(#{ident}\\.#{ident})'     ,
  'element'  :          '(#{ident})'   ,
  'pseudo class'   :    '(:#{ident})'
};
```

The scanner will work by:

- Expanding `#{}` in the macros
- Expanding `#{}` in the rules based on the expanded macros
- Escaping the backslashes
- Joining each of the patterns with `|`
- Building a global regular expression with the `RegExp` class

This will output a giant regular expression much like the ones used by Sizzle and Sly. The advantage of this approach is you can see the relationship between the tokenized output and the DOM matchers/searchers.

Processing the Giant Regular Expression

After a selector is normalised, it will be broken apart using the regular expression from the scanner. This works based on the indexes of the matched elements:

```
while  (match  = r.exec (this .selector  ))  {
  finder  = null ;

  if  (match [10 ])  {
    finder   = 'id' ;
  } else  if  (match [1])  {
    finder = 'name and id';
  } else if (match[29]) {
    finder = 'name';
  } else if (match[15]) {
    finder = 'class';
  } else if (match[20]) {
    finder = 'name and class';
  }
  this.tokens.push(new Token(match[0], finder));
}
```

Despite being obtuse, This is more efficient than looking at `match[0]` with each of the regexes in the `rules` object.

Searcher

The `Searcher` class uses the output of the `Tokenizer` to search the DOM. I've based the algorithm on how Firefox works. To recap:

> The last part of a selector is called the *key selector*. The ancestors of the key selector are analysed to determine which elements match the entire selector.

The `Searcher` is instantiated with a root element to search from and an array of tokens. The key selector is the last item in the tokens array. Each token has an identity and a *finder* — the finder is the glue between selectors and JavaScript's DOM searching methods. At the moment I've only implemented class and ID-based finders. They're stored in an object called `findMap`:

```
find = {
  byId: function(root, id) {
    return [root.getElementById(id)];
  },

  byNodeName: function(root, tagName) {
    var i, results = [], nodes = root.getElementsByTagName(tagName);
    for (i = 0; i < nodes.length; i++) {
      results.push(nodes[i]);
    }
    return results;
  },

  byClassName: function(root, className) {
    var i, results = [], nodes = root.getElementsByTagName('*');
    for (i = 0; i < nodes.length; i++) {
      if (nodes[i].className.match('\\b' + className + '\\b')) {
        results.push(nodes[i]);
      }
    }
    return results;
  }
};

findMap = {
  'id': function(root, selector) {
    selector = selector.split('#')[1];
    return find.byId(root, selector);
  },

  'name and id': function(root, selector) {
    var matches = selector.split('#'), name, id;
    name = matches[0];
    id = matches[1];
    return  filter .byAttr (find .byId (root , id), 'nodeName' , name .toUpperCase ());
  }

  // ...
};
```

The `byClassName` method is replaced when browsers support `getElementsByClassName`. The implementation I've used here is the most basic and readable one I could think of.

The `Searcher` class abstracts accessing this layer of the selector engine by implementing a `find` method:

```
Searcher .prototype .find = function (token ) {
  if (!findMap [token .finder ]) {
    throw new InvalidFinder ('Invalid finder: ' + token .finder );
  }
  return findMap [token .finder ](this .root , token .identity );
};
```

An exception is thrown when the finder does not exist. The core part of the algorithm is `matchesToken`. This is used to determine if a token matches a given node:

```
Searcher .prototype .matchesToken = function (element , token ) {
  if (!matchMap [token .finder ]) {
    throw new InvalidFinder ('Invalid matcher: ' + token .finder );
  }
  return matchMap [token .finder ](element , token .identity );
};
```

This looks a lot like the `find` method. The only difference is `matchMap` is used to check if an element matches a selector, rather than searching for nodes that match a selector.

Each token for a given list of nodes is matched using this method in `matchesAllRules`. A `while` loop is used to iterate up the tree:

```
while ((ancestor = ancestor .parentNode ) && token ) {
  if (this .matchesToken (ancestor , token )) {
    token = tokens .pop ();
  }
}
```

If there are no tokens at the end of this process then the element matches all of the tokens.

With all that in place, searching the DOM is simple:

```
Searcher .prototype .parse = function () {
  // Find all elements with the key selector
  var i, element , elements = this .find (this .key_selector ), results = [];

  // Traverse upwards from each element to see if it matches all of the rules
  for (i = 0; i < elements .length ; i++) {
    element = elements [i];
    if (this .matchesAllRules (element )) {
      results .push (element );
    }
  }
  return results ;
};
```

Each element that matches the key selector is used as a starting point. Its ancestors are analysed to see if they match all of the selector's rules.

As I said in the previous tutorial this isn't going to be the fastest selector engine ever made, but it is easy to understand and extend.

Implementing the API

Now we have all these tools for searching nodes and parsing selectors we need to knit them into a usable API. I decided to expose the tokenizer during testing — this was inspired by Sly which gives access to its parser's output.

The public methods look like this:

```
dom .tokenize   = function  (selector  )  {
  var  tokenizer   = new  Tokenizer  (selector  );
  return   tokenizer  ;
};

dom .get  = function  (selector  )  {
  var  tokens   = dom .tokenize  (selector  ). tokens  ,
       searcher   = new  Searcher  (document  ,  tokens );
  return   searcher  .parse ();
};
```

The method most people will generally use is `dom.get`, which takes a selector and returns an array of elements. The elements aren't currently wrapped in a special object like jQuery uses, but this would be a useful future improvement. I thought it would be interesting to leave this out and implement it when other parts of framework actually need it.

Tests

As I was writing the selector engine I wrote tests. Unlike other turing tests these require a browser to run — I could get around this, and it would be nice if the selector engine could search arbitrary XML, but to keep it simple I've kept this constraint.

I started developing the selector engine by writing this test:

```
 given ('a selector to search for'     , function  ()  {
    should  ('find with id'   ,  turing  .dom .get ('#dom-test'   )[ 0]. id). isEqual  ('dom-test'  );
```

Once that was done I moved on to searching for classes, then combinations of tag names and classes.

I've built the framework with this test-first approach, and I highly recommend it for your own projects.

onReady

An `onReady` handler watches for the DOM to finish loading, rather than the entire document (with all of the related images and other assets). This is useful because it gives the illusion of JavaScript-related code being instantly available. If this wasn't done, JavaScript code could be evaluated after the user has started interacting with the document.

Most jQuery users use this feature without realising it's there:

```
$(document  ). ready (function  ()  {
  // Let the fun begin
});
```

Here’s the core of what makes this happen:

```
bindReady  :  function  ()  {
  if  (  readyBound    )  {
    return  ;
  }

  readyBound    =  true  ;

  // Catch cases where $(document).ready() is called after the
  // browser event has already occurred.
  if  (  document  .readyState    ===   "complete"    )  {
    return   jQuery  .ready  ();
  }

  // Mozilla, Opera and webkit nightlies currently support this event
  if  (  document  .addEventListener     )  {
    // Use the handy event callback
    document  .addEventListener    (  "DOMContentLoaded"    ,  DOMContentLoaded    ,  false   );

    // A fallback to window.onload, that will always work
    window  .addEventListener    (  "load"  ,  jQuery  .ready  ,  false   );

  // If IE event model is used
  }  else  if  (  document  .attachEvent    )  {
    // ensure firing before onload,
    // maybe late but safe also for iframes
    document  .attachEvent   ("onreadystatechange"    ,  DOMContentLoaded    );

    // A fallback to window.onload, that will always work
    window  .attachEvent   (  "onload"  ,  jQuery  .ready   );

    // If IE and not a frame
    // continually check to see if the document is ready
    var  toplevel    =  false  ;

    try  {
      toplevel    =  window  .frameElement    ==  null  ;
    }  catch  (e)  {}

    if  (  document  .documentElement    .doScroll    &&  toplevel    )  {
      doScrollCheck    ();
    }
  }
}
```

Prototype and other frameworks have similar code. It’s not surprising that Prototype’s DOM loaded handling references Resig and other prominent developers (Dan Webb, Matthias Miller, Dean Edwards, John Resig, and Diego Perini). `jQuery.ready` gets called through either a modern `DOMContentLoaded` event, or `onload` events.

I like the way Prototype fires a custom event when the document is ready. Prototype uses a colon to denote a custom event, because these have to be handled differently by IE — `element.fireEvent('something else', event)` causes an argument error. I tried to duplicate that in Turing, but it’d take a fair amount of work to adapt Turing’s current event handling so I left it out for now and used an array of callbacks instead.

Setting up multiple observer callbacks will work:

```
<!DOCTYPE html>
<html>
```

```
<head>
  <style>p  {  color :red ;  }</style>
  <script  src= "http://code.jquery.com/jquery-latest.min.js"        ></script>
  <script>
  $(document  ). ready (function  ()  {
    $("#a" ). text ("A." );
  });

  $(document  ). ready (function  ()  {
    $("#b" ). text ("B." );
  });

  $(document  ). ready (function  ()  {
    $("#c" ). text ("C." );
  });
  </script>

</head>
<body>
  <p  id= "a" ></p>
  <p  id= "b" ></p>
  <p  id= "c" ></p>
</body>
</html>
```

Our API

I'm shooting for this:

```
turing .events .ready (function  ()  {
  // The DOM is ready, but some other stuff might be
});
```

Implementing "onready"

I've based the current code on jQuery. It's just enough to work cross-browser and do what I need it to do. It's all handled by private methods and variables. The last thing to get called is `ready()`:

```
function  ready ()  {
  if  (!isReady )  {
    // Make sure body exists
    if  (!document  .body )  {
      return  setTimeout  (ready ,  13);
    }

    isReady  = true ;

    for  (var  i  in  readyCallbacks  )  {
      readyCallbacks  [i]();
    }

    readyCallbacks   = null ;
  }
}
```

This is just a tiny snippet, but to summarise the rest of the code:

1. The public method, `turing.events.ready` calls `bindOnReady`
2. If `bindOnReady` has already been called once, it will return
3. This method sets up `DOMContentLoaded` as an event, and will fall over to simply call `ready()`

4. The “doScroll check” is used for IE through `DOMReadyScrollCheck`, which also calls `ready()` when it’s done
5. `setTimeout` and recursive calls to `DOMReadyScrollCheck` make this happen

The `isReady` variable is bound to the closure for `turing.events`, so it’s safely tucked away where we need it. The rest of the code is similar to jQuery’s — after going through the tickets referenced in the code comments relating to IE problems, I didn’t have enough time to create my own truly original version.

Conclusion

I hope you’ve learned why selector engines are so complex and why we need them. Granted, it’s mostly due to browser bugs that don’t need to exist, but browsers and standards both evolve out of step, so sometimes it’s our job to patch the gaps while they catch up.

While it’s true that Sizzle is the mother of all selector engines, I think implementing a small, workable one will be a worthwhile project.

Events

Introduction

This chapter will look at how events work, event handler implementations in various frameworks, and event handler API designs.

Basics

Events and JavaScript are closely related — can you imagine writing scripts for web pages that don't respond to user interaction? That means that as soon as JavaScript appeared, events did. Early event handlers were written inline, like this:

```
<a  href ="/"  onclick  ="alert('Hello World!')"     >
```

You've probably seen this type of inline event handler before. This came from Netscape originally, and due to the popularity of early versions of Netscape, Microsoft implemented a compatible system.

This can be written in JavaScript like this:

```
// assume 'element' is the previous link minus the onclick
element  .onclick   =  function  ()  {  alert ('Hello World!'   );  };
```

Accessing the Event

An event handler can access an event like this:

```
function   handler  (event )  {
if  (!event )  var  event   =  window  .event ;
}
```

`window.event` is a Microsoft property for the last event. I always felt like this was dangerous and could lead to to a clobbered value, but as JavaScript is single-threaded it's safe enough that most frameworks depend on it.

jQuery does it like this:

```
handle  :  function   (  event   )  {
  var  all ,  handlers   ,  namespaces   ,  namespace_sort    =  [] ,  namespace_re   ,  events  ,  args   =  jQuery  .makeArra
  event   =  args [0]  =  jQuery  .event  .fix (  event   ||  window  .event   );
```

Stopping Events

I used to get default actions and bubbling confused, because I thought stopping an event just meant everything stopped. These two things are different.

Default Action

Returning `false` from an event handler prevents the default action:

```
element  .onclick   =  function  ()  {  alert ('Hello World!'   );  return   false ;  };
```

Now the link won't be followed.

Capturing and Bubbling

Attaching an `onClick` to two elements, where one is the ancestor of the other, makes it difficult to tell which element should take precedence. Different browsers do it different ways. However, fortunately it's very unusual to actually care about this — in most cases we just need to stop the event.

```
function  handler (event ) {
if (!event ) var event  = window .event ;
event .cancelBubble   = true ;
if (event .stopPropagation   ) event .stopPropagation   ();
}
```

As far as I know, only IE uses `cancelBubble`, but it's safe to set it in browsers that don't use it. `stopPropagation` is used by most browsers.

jQuery's implementation in event.js is similar to the above, and most frameworks are broadly similar.

Multiple Handlers

If events were this simple we'd barely need frameworks to help us. One of the things that makes things less simple is attaching multiple events:

```
element .onclick  = function  () { alert ('Hello World!'  ); return  false ; };
element .onclick  = function  () { alert ('This was the best example I could think of'        ; return  false ;
```

This example overwrites the `onClick` handler, rather than appending another one.

The solution isn't as simple as wrapping functions within functions, because people might want to remove event handlers in the future.

Framework APIs

The job of an event framework is to make all of these things easy and cross-browser. Most do the following:

- Normalise event names, so `onClick` becomes `click`
- Easy event registration and removal
- Simple cross-browser access to the event object in handlers
- Mask the complexities of event bubbling
- Provide cross-browser access to keyboard and mouse interaction
- Patch browser incompetency like IE memory leaks

jQuery

Interestingly, jQuery's event handling approach is to make the event object behave like the W3C standards. The reason for this is that Microsoft failed to implement a compatible API.

jQuery wraps methods into its internal DOM list objects, so the API feels very easy to use:

```
$('a').click (function  (event ) {
  alert ('A link has been clicked'    );
});
```

This adds the `click` handler to every link.

- The Event's element is in `event.target`

- The current event within the bubbling phase is in `event.currentTarget` — also found in the `this` object in the function
- The default action can be prevented with `event.preventDefault()`
- Bubbling can be stopped with `event.stopPropagation();`
- Events can be removed with `$('a').unbind('click');`
- Events can be fired with `$('a').trigger('click');`

Prototype

Prototype's event handling is closely linked to its core DOM `Element` class. Events are registered like this:

```
$('id').observe('click', function(event) {
  var element = Event.element(event);
});
```

- The Event's element can be accessed with `Event.element(event)` or `event.element();`
- Events are stopped with `Event.stop()`
- Events can be removed with `Event.stopObserving(element, eventName, handler);`
- Events can be fired with `Event.fire(element)`

Glow

Glow's event handling is found in glow.events. Events are registered with `addListener`:

```
glow.events.addListener('a', 'click', function(event) {
  alert('Hello World!');
});
```

- The Event's element is in `event.source`
- The default action can be prevented by returning `false`
- Events can be removed with `glow.events.removeListener(handler)` where `handler` is the returned value from `glow.events.addListener`
- Events can be fired with `glow.events.fire()`

Glow's jQuery influence is evident here.

Dojo

Rather than ironing out the problems in browser W3C event handling supporting, dojo uses a class-method-based system like Prototype, but with a different approach. Dojo's API is based around *connections* between functions. Registering a handler looks like this:

```
dojo.connect(dojo.byId('a#hello'), 'onclick', function(event) {
  alert('Hello World!');
});
```

Notice that the event names aren't mapped. Like the other frameworks, the `event` object is normalised.

- The Event's element is in `event.target`
- The current event within the bubbling phase is in `event.currentTarget` — this is also `this` in the function
- The default action can be prevented with `event.preventDefault()`
- Bubbling can be stopped with `event.stopPropagation();`
- Events can be removed with `dojo.disconnect()`

Summary

Out of all these frameworks, jQuery's event handling is the most fascinating. It makes a familiar W3C event system available to all browsers, carefully namespaced, and completely takes over event bubbling to achieve this. This is partially because Microsoft's API has major problems with bubbling.

Building something in between jQuery and Glow is suitable for Turing — I don't want to worry about bubbling too much, but we will need a system that will cope with multiple handlers on the same event and the removal and firing of handlers.

Goals

After looking at how events work and how popular frameworks are built, the goals for our events code are:

- Normalise event names, so `onClick` becomes `click`
- Easy event registration and removal
- Simple cross-browser access to the event object in handlers
- Mask the complexities of event bubbling
- Provide cross-browser access to keyboard and mouse interaction
- Patch browser incompetency like IE memory leaks

W3C and Microsoft

The Level 2 Events Specification tried to encourage browser developers to provide a better API, but in the end wasn't supported by Microsoft. From my experiences of framework-less JavaScript I can tell you that this model is pretty sound.

The quest for browser support using this API has resulted in lots of similar event handling implementations. Most conditionally use the W3C or Microsoft APIs.

W3C Event Handling

Adding an event handler to an element looks like this:

```
element.addEventListener('click', function() { }, false);
```

Type is W3C's terminology for event names, like 'click'. The third parameter determines if capturing should be initiated. We're not going to worry about capturing here. The event handler is passed an `event` object, which has a `target` property.

Events can be removed with `removeEventListener()`, with the same parameters. It's important that the callback matches the one the event was registered with.

Events can be fired programatically with `dispatchEvent()`.

Microsoft

Before getting disgruntled about Microsoft breaking the universe again, their implementation almost stands up:

```
var handler = function() { };
element.attachEvent('onclick', handler);
element.detachEvent('onclick', handler);
```

```
// Firing events
event = document.createEventObject();
return element.fireEvent('on' + type, event)
```

This is mostly similar to W3C's recommended API, except 'on' is used for event type names.

The two main issues with Microsoft's implementation are memory leaks and the lack of a `target` parameter. Most frameworks handle memory leaks by caching events and using `onunload` to register a handler which clears events.

As for `target`, jQuery's `fix` method maps IE's proprietary srcElement property, and Prototype does something similar when it extends the `event` object:

```
Object.extend(event, {
  target: event.srcElement || element,
  relatedTarget: _relatedTarget(event),
  pageX: pointer.x,
  pageY: pointer.y
});
```

Capabilities and Callbacks

Our event handling code would be a lot simpler without Microsoft's API, but it's not a huge problem for the most part. Capability detection is simple in this case because there's no middle-ground — browsers either use W3C's implementation or they're IE. Here's an example:

```
if (element.addEventListener) {
  element.addEventListener(type, responder, false);
} else if (element.attachEvent) {
  element.attachEvent('on' + type, responder);
}
```

The same approach can be repeated for removing and firing events. The messy part is that the event handlers need to be wrapped in an anonymous function so the framework can correct the `target` property. The process looks like this:

1. The event is set up by the framework: `turing.events.add(element, type, handler)`
2. The handler is wrapped in a callback that can fix browser differences
3. The event object is passed to the original event handler
4. When an event is removed, the framework matches the passed in event handler with ones in a "registry", then pulls out the wrapped handler

Both jQuery and Prototype use a cache to resolve IE's memory leaks. This cache can also be used to help remove events.

Valid Elements

Before adding an event, it's useful to check that the supplied element is valid. This might sound strange, but it makes sense in projects where events are dynamically generated (and also with weird browser behaviour).

The `nodeType` is checked to make sure it's not a text node or comment node:

```
function isValidElement(element) {
  return element.nodeType !== 3 && element.nodeType !== 8;
}
```

I got this idea from jQuery's source.

API Design

At the moment the Turing API reflects W3C's events:

- Add an event: `turing.events.add(element, type, callback)`
- Remove an event: `turing.events.remove(element, type, callback)`
- Fire: `turing.events.fire(element, type)`
- An `event` object is passed to your callback with the `target` property fixed

Tests

Like previous parts of this project, I started out with tests. I'm still not quite happy with the tests though, but they run in IE6, 7, 8, Chrome, Firefox and Safari.

```
var element = turing.dom.get('#events-test a')[0],
    check = 0,
    callback = function(e) { check++; return false; };

should('add onclick', function() {
  check = 0;
  turing.events.add(element, 'click', callback);
  turing.events.fire(element, 'click');
  turing.events.remove(element, 'click', callback);
  return check;
}).equals(1);
```

I use a locally-bound variable called `check` which counts how many time an event has fired using the event's handler.

The tests also ensure that attaching global handlers work (on `document`).

Stopping Events

Once an event has been triggered it can propagate to other elements — this is known as *event bubbling*. To understand this, try to think about what happens if you have a container element and attach an event to an element inside it. When the event is triggered, which elements should receive the event?

We often don't want to propagate events at all. In addition, some elements have *default actions* — a good example is how a link tag's default action makes the browser follow the link.

Prototype's Event.stop() method simplifies event management by cancelling event propagation and default actions. We generally want to do both at the same time.

jQuery models the W3C's Document Object Model Events spec, providing lots of methods on the event object itself:

- `event.preventDefault()`: Stop the default action of the event from being triggered
- `event.stopPropagation()`: Prevents the event from bubbling up the DOM tree, preventing any parent handlers from being notified of the event
- `event.stopImmediatePropagation()`: Keeps the rest of the handlers from being executed and prevents the event from bubbling up the DOM tree

Our Stop API

I’ve modelled Turing’s API on jQuery, with the addition of `stop()`. The reason I like jQuery’s approach is it creates a cross-browser W3C API, which may future-proof the library.

Event objects are extended with:

- `event.stop()` – Prevents the default handler and bubbling
- `event.preventDefault()` – Prevents default handler
- `event.stopPropagation()` – Stops the event propagating

Usage is best illustrated with a test from test/events_test.js

```
should ('stop' , function () {
  var callback = function (event ) { event .stop (); };
  turing .events .add (turing .dom .get ('#link2' )[ 0], 'click' , callback );
  // ...
```

The Implementation

I’ve created a private function to extend and fix event objects. This essentially patches IE and adds `stop()`:

```
function stop (event ) {
  event .preventDefault (event );
  event .stopPropagation (event );
}

function fix (event , element ) {
  if (!event ) var event = window .event ;

  event .stop = function () { stop (event ); };

  if (typeof event .target === 'undefined' )
    event .target = event .srcElement || element ;

  if (!event .preventDefault )
    event .preventDefault = function () { event .returnValue = false ; };

  if (!event .stopPropagation )
    event .stopPropagation = function () { event .cancelBubble = true ; };

  return event ;
}
```

Other Browser Fixes

Most frameworks patch other browser inconsistencies as well. Keyboard and mouse handling in particular are problematic.

jQuery corrects the following:

- Safari’s handling of text nodes
- Missing values for `event.pageX/Y`
- Key events get `event.which` and `event.metaKey` is corrected
- `event.which` is also added for mouse button index

Prototype also has similar corrections:

```
var _isButton ;
if (Prototype .Browser .IE ) {
  // IE doesn't map left/right/middle the same way.
  var buttonMap = { 0: 1, 1: 4, 2: 2 };
  _isButton = function (event , code ) {
    return event .button === buttonMap [code ];
  };
} else if (Prototype .Browser .WebKit ) {
  // In Safari we have to account for when the user holds down
  // the "meta" key.
  _isButton = function (event , code ) {
    switch (code ) {
      case 0: return event .which == 1 && !event .metaKey ;
      case 1: return event .which == 1 && event .metaKey ;
      default : return false ;
    }
  };
} else {
  _isButton = function (event , code ) {
    return event .which ? (event .which === code + 1) : (event .button === code );
  };
}
```

You can also find similar patching in MooTools:

```
if (type .test (/key/ )){
  var code = event .which || event .keyCode ;
  var key = Event .Keys .keyOf (code );
  if (type == 'keydown' ){
    var fKey = code - 111 ;
    if (fKey > 0 && fKey < 13) key = 'f' + fKey ;
  }
  key = key || String .fromCharCode (code ).toLowerCase ();
} else if (type .match (/(click|mouse|menu)/i )){
  doc = (!doc .compatMode || doc .compatMode == 'CSS1Compat' ) ? doc .html : doc .body ;
  var page = {
    x: event .pageX || event .clientX + doc .scrollLeft ,
    y: event .pageY || event .clientY + doc .scrollTop
  };
  var client = {
    x: (event .pageX ) ? event .pageX - win .pageXOffset : event .clientX ,
    y: (event .pageY ) ? event .pageY - win .pageYOffset : event .clientY
  };
  if (type .match (/DOMMouseScroll|mousewheel/ )){
    var wheel = (event .wheelDelta ) ? event .wheelDelta / 120 : -(event .detail || 0) / 3;
  }
```

Event Delegation

Event delegation is where an event is attached to a parent element (or the whole document), then selectors are used to determine if a handler should run.

Consider the following markup:

```
<ul id="navigation" >
  <li><a href="#page_1" >Page 1 </a></li>
  <li><a href="#page_2" >Page 2 </a></li>
  <li><a href="#page_3" >Page 3 </a></li>
</ul>
```

An event handler could be attached to `#navigation` and watch for clicks on the links. The advantage of this is new links can be added (perhaps through Ajax) and the event handler will still work. It also uses less events when compared to attaching events to each link.

This approach was a revelation 5 or 6 years ago, but it's now considered best practice for a wide range of applications.

In the Wild

jQuery offers two methods for dealing with this:

```
$('#navigation a').live('click', function() {
  // Handler
});

$('#navigation').delegate('a', 'click', function() {
  // Handler
});
```

The live and delegate methods are very similar and share the same code underneath.

The way this used to work in Prototype (and other Prototype-influenced frameworks) was like this:

```
$('navigation').observe('click', function(event) {
  var element = event.findElement('a');
  if (element) {
    // Handler
  }
});
```

As you can see, jQuery's API saves a little bit of boilerplate code. Prototype 1.7 introduced `on` which can be used like this:

```
$('navigation').on('click', 'a', function(event, element) {
  // Handler
});
```

This is more like jQuery's `delegate` method.

Underneath

jQuery relies on several things to handle delegation. The main method is `liveHandler`, in event.js:

1. `jQuery.data` is used to track event handler objects in the current context
2. `closest` is used to find elements based on the event's `target` and `currentTarget`, and the selectors found in the previous step
3. Each matching element is compared against the selectors to generate a list of matching elements
4. This element list is then used to run the user-supplied event handlers
5. If the handler returns false or propagation has been stopped, false will be returned

The Prototype approach is simpler — it loses some of the flexibility of the jQuery approach — but might be a better candidate to base our framework code on. Let's target the original Prototype style (that used `findElement`) and make it generic.

```
function findElement(event, expression) {
  var element = Event.element(event);
  if (!expression) return element;
  while (element) {
    if (Prototype .Selector .match (element , expression )) {
      return  Element .extend (element );
    }
    element  = element .parentNode  ;
  }
}
```

This code loops through each element, going up the DOM, until an element is found that matches the selector. Turing already has some internal DOM methods we can use to implement this feature.

dom.findElement

This method will make dealing with event delegation much easier:

```
$('navigation' ).observe ('click' , function (event ) {
  var element  = event .findElement  ('a' );
  if (element ) {
    // Handler
  }
});
```

This Prototype code simply uses the event’s target element to see if it matches a selector. Turing’s delegation API could wrap this up a method with a signature like this:

```
turing .events .delegate (document , selector , 'click' , function (e) {
  // Handler
});
```

The body of `delegate` will look a bit like the Prototype example above.

There’s a few obstacles to building `findElement` with Turing’s event library as it stands though. A few months ago we built a class called `Searcher` that can recurse through the DOM to match tokenized CSS selectors.

The `Searcher` class could be reused to implement `findElement`, the `matchesAllRules` method in particular is of interest.

Tests

I’d like the following tests to pass (events.js):

```
given ('a delegate handler'   , function () {
  var clicks  = 0;
  turing .events .delegate (document , '#events-test a'   , 'click' , function (e) {
    clicks ++;
  });

  should ('run the handler when the right selector is matched'          , function () {
    turing .events .fire (turing .dom .get ('#events-test a'   )[0], 'click' );
    return  clicks ;
  }).equals (1);

  should ('only run when expected'    , function () {
    turing .events .fire (turing .dom .get ('p')[0], 'click' );
    return  clicks ;
  }).equals (1);
});
```

To get there, we need `findElement` and corresponding tests (dom.js):

```
given ('a nested element'    , function  ()  {
  var  element   = turing .dom .get ('#dom-test a.link'    )[0];
  should  ('find elements with the right selector'        , function  ()  {
    return   turing .dom .findElement   (element  ,  '#dom-test a.link'    , document  );
  }). equals  (element  );

  should  ('not find elements with the wrong selector'         ,function  ()  {
    return   turing .dom .findElement   (turing  .dom .get ('#dom-test .example1 p'     )[0],  'a.link'  ,  document  )
  }). equals  (undefined  );
});
```

This test depends on the markup in dom_test.html.

Adapting the Searcher Class

After I looked at `matchesAllRules`, in turing.dom.js I realised it shouldn't be too hard to make it more generic. It previously took an element and searched its ancestors, but we need to include the current element in the search.

To understand why, consider how `findElement` should work (this is simplified code):

```
dom .findElement    = function  (element  ,  selector  ,  root )  {
  while   (element  )  {
    if  (matchesAllRules   (selector  ,  element  ))  {
      // We've found it!
      return   element  ;
    }
    // Else try again with the parent
    element   = element  .parentNode  ;
  }
};
```

All I had to do was refactor code that relies on `matchesAllRules` to pass an element's `parentNode` instead of the element itself.

The start of the `matchesAllRules` method now looks slightly different:

```
Searcher  .prototype  .matchesAllRules    = function  (element  )  {
  var  tokens   = this .tokens  .slice () ,  token   = tokens  .pop () ,
      matchFound    = false ;
```

The code that refers to the `ancestor` element has been removed and the `element` argument is used instead.

The Event Delegation Method

We need to wrap the user's event handler with one that checks the element is one we're interested in, and other than that it's standard Turing event handling:

```
if  (turing  .dom  !==  'undefined'   )  {
  events  .delegate   = function  (element  ,  selector  ,  type ,  handler  )  {
    return   events  .add (element  ,  type ,  function  (event  )  {
      var  matches   = turing  .dom .findElement   (event  .target  ,  selector  ,  event .currentTarget   );
      if  (matches  )  {
        handler  (event );
      }
    });
  };
}
```

This code checks to see if `dom` is available because we don't want interdependence between the modules. Then it sets up a standard event handler and uses `findElement` to ensure the event is one we're interested in.

NodeList, Collections and Arrays

NodeList and Collections

DOM Level 1 defines `NodeList` as an abstraction of ordered collections of nodes. The specification is kept simple to avoid constraining the underlying implementation. That means collections of DOM elements can't be directly manipulated like an array, although it's trivial to iterate over each element:

```
// Get the elements
var elements = document.querySelectorAll('p');

// Iterate over each element with a simple for loop
for (var i = 0; i < elements.length; i++) {
  console.log(elements[i]);
}
```

To see why `NodeList` doesn't do what we want, you'll notice `Array.prototype` methods are missing:

```
document.querySelectorAll('p').push
// returns undefined
```

Ouch!

Another interesting point about `NodeList` is it's an *ordered* collection. From DOM Level 1 Core:

> `getElementsByTagName` Returns a NodeList of all descendant elements with a given tag name, in the order in which they would be encountered in a preorder traversal of the Element tree.

Over at the Mozilla NodeList documentation, they have this to say:

> This is a commonly used type which is a collection of nodes returned by getElementsByTagName, getElementsByTagNameNS, and Node.childNodes. The list is live, so changes to it internally or externally will cause the items they reference to be updated as well. Unlike NamedNodeMap, NodeList maintains a particular order (document order). The nodes in a NodeList are indexed starting with zero, similarly to JavaScript arrays, but a NodeList is not an array.

This isn't just what Mozilla implementations do, technically all browsers should. This is from the specifications:

> NodeLists and NamedNodeMaps in the DOM are "live", that is, changes to the underlying document structure are reflected in all relevant NodeLists and NamedNodeMaps.

Mozilla's documentation also points out a gotcha that I've run into before:

> Don't be tempted to use `for...in` or `for each...in` to enumerate the items in the list, since that will also enumerate the length and item properties of the NodeList and cause errors if your script assumes it only has to deal with element objects.

Converting NodeList into an Array

The simplest approach is probably the best:

```
function  toArray (collection  ) {
  var  results   = [];
  for  (var  i  =  0;  i  <  collection  .length ;  i++) {
    results .push (collection  [i]);
  }
  return  results ;
}
```

The major downside of this is the results will no-longer be *live* — the original NodeList is a reference to a set of objects rather than a fixed result set.

In the Wild

jQuery has a method called `makeArray`:

```
makeArray  : function  ( array ,  results   ) {
  var  ret  =  results   ||  [];

  if  (  array   !=  null  )  {
    // The window, strings (and functions) also have 'length'
    // The extra typeof function check is to prevent crashes
    // in Safari 2 (See: #3039)
    if  (  array .length   ==  null   ||  typeof   array   ===  "string"   ||  jQuery .isFunction   (array ) ||  (typeof
      push .call (  ret ,  array   );
    }  else  {
      jQuery .merge (  ret ,  array   );
    }
  }

  return  ret ;
}
```

In this code, `push` refers to `Array.prototype.push`.

When Prototype uses `querySelectorAll`, it wraps the output in `$A()` and uses `.map(Element.extend)` to make each element a Prototype `Element`. This is similar to the above, with the exception of Prototype extending each element.

Some other frameworks wrap the results in their own `NodeList` class, rather than converting them to an array.

Implementation

The `toArray` function described above has been added to turing.core.js in the form of `turing.toArray` and added to `turing.dom.get`.

References

- Document Object Model (Core) Level 1
- Document Object Model Events
- Mozilla NodeList documentation
- Node.nodeType

- Understanding and Solving Internet Explorer Leak Patterns
- srcElement Property
- createEventObject Method
- QuirksMode on registration models
- addEvent() recoding contest
- Prototype’s event.js
- jquery’s event.js

Ajax

XMLHttpRequest

All XMLHttpRequest really does is allows browsers to send messages *back* to the server. Combined with DOM manipulation, this allows us to update or replace parts of a page. This simple API marked a radical change in web application development, and rapidly became commonplace.

Requests must adhere to the *same origin policy*. This means a request has to call a server at a given domain. This limitation can be side-stepped by using dynamically inserted script tags, image tags, and iframes.

History

XMLHttpRequest is another technology that one player in the browser war pioneered, while the others had to catch up. What's interesting about XMLHttpRequest is Microsoft created the concept behind it — not Mozilla or the W3C. Although ActiveX was used, equivalent functionality shipped with Internet Explorer 5.0 in 1999. It wasn't until 2006 that the W3C published a working draft for XMLHttpRequest itself.

Request Objects

In this series of tutorials I'm going to focus on XMLHttpRequest as well as other network-related techniques. Following Glow's lead, I'll use the namespace `turing.net`.

Frameworks like jQuery make XMLHttpRequest look incredibly simple, but there's actually a fair bit going on behind the scenes:

1. Differences between Microsoft's implementation and W3C have to be dealt with
2. Request headers must be set for the type of data and HTTP methpd
3. State changes must be handled through callbacks
4. Browser bugs must be compensated for

Creating a request object looks like this:

```
// W3C compliant
new XMLHttpRequest();
// Microsoft
new ActiveXObject("Msxml2.XMLHTTP.6.0");
new ActiveXObject("Msxml2.XMLHTTP.3.0");
new ActiveXObject('Msxml2.XMLHTTP');
```

Internet Explorer has different versions of MSXML — these are the preferred versions based on Using the right version of MSXML in Internet Explorer.

jQuery's implementation looks like this:

```
xhr: window.XMLHttpRequest && (window.location.protocol !== "file:" || !window.ActiveXObject) ?
  function() {
    return new window.XMLHttpRequest();
  } :
  function() {
    try {
      return new window.ActiveXObject("Microsoft.XMLHTTP");
    } catch(e) {}
  },
```

This code uses a ternary operator to find a valid transport. It also has a note about preventing IE7 from using XMLHttpRequest, because it can't load local files.

More explicitly, finding the right request object looks like this:

```
function  xhr () {
  if (typeof  XMLHttpRequest   !== 'undefined'  && (window .location  .protocol  !== 'file:'  || !window .A
    return  new XMLHttpRequest  ();
  } else {
    try {
      return  new ActiveXObject  ('Msxml2.XMLHTTP.6.0'   );
    } catch (e) { }
    try {
      return  new ActiveXObject  ('Msxml2.XMLHTTP.3.0'   );
    } catch (e) { }
    try {
      return  new ActiveXObject  ('Msxml2.XMLHTTP'  );
    } catch (e) { }
  }
  return  false ;
}
```

Sending Requests

There are three main parts to sending a request:

1. Set a callback for `onreadystatechange`
2. Call `open` on the request object
3. Set the request headers
4. Call `send` on the object

The `onreadystatechange` callback can be used to call user-supplied callbacks for request success and failure. The following states may trigger this callback:

- 0: *Uninitialized*: open has not been called
- 1: *Loading*: `send` has not been called
- 2: *Loaded*: `send` has been called, headers and status are available
- 3: *Interactive*: Downloading, `responseText` holds the partial data
- 4: *Completed*: Request finished

The `open` function is used to initialize the HTTP method, set the URL, and determine if the request should be asynchronous. Request headers can be set with `request.setRequestHeader(header, value)`. The post body can be set with `send(postBody)`.

This simple implementation uses the previously defined `xhr` function to send requests:

```
function  ajax (url , options ) {
  function  successfulRequest   (request ) {
    return  (request .status  >= 200  && request .status  < 300 ) ||
        request .status  == 304 ||
(request .status  == 0 && request .responseText  );
  }

  function  respondToReadyState   (readyState  ) {
    if (request .readyState  == 4 ) {
      if (successfulRequest   (request )) {
        if (options .success ) {
          options .success (request );
        }
      } else  {
        if  (options  .failure  )  {
          options  .failure  ();
        }
      }
    }
  }

  function   setHeaders  ()  {
    var  headers   = {
      'Accept'  :  'text/javascript, text/html, application/xml, text/xml, */*'
    };

    for  (var  name  in  headers  )  {
      request  .setRequestHeader   (name ,  headers  [name ]);
    }
  }

  var  request   = xhr ();
  if  (typeof   options   ===  'undefined'   )  {
    options   = {};
  }

  options  .method   = options  .method   ? options  .method  .toLowerCase   ()  :  'get' ;
  options  .asynchronous    = options  .asynchronous    ||  true ;
  options  .postBody   = options  .postBody   ||  '' ;

  request  .onreadystatechange     = respondToReadyState    ;
  request  .open (options  .method  ,  url ,  options  .asynchronous   );
  setHeaders  ();
  request  .send (options  .postBody  );
}
```

Popular APIs

jQuery uses methods like `get` and `post`. These are shorthands for the `ajax` function:

```
$.ajax ({
  url :  url ,
  data :  data ,
  success  :  success  ,
  dataType  :  dataType
});
```

Prototype uses classes — `Ajax.Request` is the main one. Typical usage looks like this:

```
new  Ajax .Request  ('/your/url'  ,  {
  onSuccess  :  function  (response  )  {
    // Handle the response content...
  }
});
```

Glow framework 1.7 is interesting because it fires a custom event when an event succeeds or fails:

```
if  (response  .wasSuccessful   )  {
  events  .fire (request  ,  "load"  ,  response  );
}  else  {
  events  .fire (request  ,  "error"  ,  response  );
}
```

Putting it Together

Using the examples above, I've built `turing.net` which implements `get`, `post`, and `ajax`:

```
turing .net .get ('/example'  , {  success  :  function  (r)  {  alert (r.responseText   );  }  });
```

It's on GitHub now: turing.net.js

Cross-Domain Requests

Cross-domain requests are useful because they can be used to fetch data from services like Twitter. This is now a popular technique, despite feeling clunky to implement. This is another case where lack of browser features can be patched by JavaScript.

Implementations in the Wild

The Glow framework has `glow.net.xDomainGet` and `glow.net.loadScript`. These are similar, but place different requirements on the server. What `loadScript` does is called JSONP. jQuery also implements JSONP, and has this to say in the documentation:

> The jsonp type appends a query string parameter of callback=? to the URL. The server should prepend the JSON data with the callback name to form a valid JSONP response. We can specify a parameter name other than callback with the jsonp option to $.ajax().

A JSONP URL looks like this:

```
http://feeds.delicious.com/v1/json/alex_young/javascript?callback=parseJSON
```

The reason a callback is specified in the URL is the cross-domain Ajax works by loading remote JavaScript by inserting `script` tags, and then interpreting the results. The term JSONP comes from *JSON with Padding*, and originated in Remote JSON – JSONP by Bob Ippolito.

The Algorithm

JSONP works like this:

1. The framework transparently creates a callback for this specific request
2. A script tag is inserted into a document. This is of course invisible to the user
3. The `src` attribute is set to `http://example.com/json?callback=jsonpCallback,`
4. The server generates a response
5. The server wraps its JSON response like this: `jsonpCallback({ ... })`
6. Once the script has loaded, the callback is called, and the client code can process the JSON accordingly
7. Finally, the callback and `script` tags are removed

As you can see, servers have to be compliant with this technique — you can't use it to fetch arbitrary JSON or XML.

API Design

I've based this on the Glow framework, and the option names are consistent with the other `turing.net` code:

```
turing .net .jsonp ('http://feeds.delicious.com/v1/json/alex_young/javascript?callback={callback}' ,
  success : function (json ) {
    console .log (json );
  }
});
```

The `{callback}` string must be specified — it gets replaced by the framework transparently.

Implementation

To create a callback, no clever meta-programming is required. It's sufficient to create a function and assign as a property on `window`:

```
methodName = '__turing_jsonp_' + parseInt (new Date (). getTime ());
window [methodName ] = function () { /* callback */ };
```

A `script` tag is created and destroyed as well:

```
scriptTag = document .createElement ('script' );
scriptTag .id = methodName ;

// Replacing the request URL with our internal callback wrapper
scriptTag .src = url .replace ('{callback}' , methodName );
document .body .appendChild (scriptTag );

// The callback should delete the script tag after the client's callback has completed
document .body .removeChild (scriptTag )
```

That's practically all there is to it. You can see the real implementation in turing.net.js by searching for `JSONPCallback`.

Conclusion

Although JSONP requires server support, a wide variety of interesting services support it. This simple technique has made cross-domain requests popular — how many blogs and sites now feature live Twitter feeds written with pure JavaScript?

References

- Glow
- jQuery/ajax.js
- The history of XMLHttpRequest
- Using the right version of MSXML in Internet Explorer
- XMLHTTP notes: readyState and the events

Animations

JavaScript Animation

JavaScript animation libraries are usually comprised of the following elements:

- Methods to work with CSS properties and animate them
- A queuing system, for scheduling animations, and animating each “frame”
- CSS colour parsing
- Helpers that make common web effects easy to use alongside events

Animation Frameworks

One of the first popular libraries that addressed animation was script.aculo.us. The effects.js script offered many pre-baked effects that developers wanted to use on their sites, and transition effects like `Effect.Highlight`, `Effect.Appear` and `Effect.BlindDown` quickly became popular. These effects are built using some JavaScript logic to manipulate CSS properties.

The script.aculo.us API is based around instantiated objects:

```
new  Effect .EffectName  (element  ,  required   parameters   ,  [options ]);
```

Most effects take a `duration`, `from` and `to` parameter:

```
new  Effect .Opacity  ('element'  ,  {
  duration  :  2.0 ,
  transition   :  Effect .Transitions   .linear  ,
  from :  1.0 ,
  to :  0.5
});
```

The `transition` option refers to a control function that determines the rate of change.

MooTools has a similar API in its Fx module:

```
new  Fx .Reveal  ($('element'  ),  {  duration  :  500 ,  mode :  'horizontal'    });
```

Shortcuts can be added to the `Element` class as well:

```
$('element'  ). reveal  ({  duration  :  500 ,  mode :  'horizontal'    });
```

jQuery provides another animation API with helpers and CSS property manipulation. jQuery UI and other plugins build or extend this functionality. The main method is animate which accepts properties to animate, duration, easing function and a termination callback. Most tasks can be completed with the helpers.

The nice thing about jQuery’s animation API is it’s very easy to create sequences of animations — just chain a list of calls:

```
$('#element'  ). slideUp  (300 ). delay  (800 ). fadeIn  (400 );
```

jQuery builds a queue to create these animation sequences. The queue documentation has an example that demonstrates how a queue is built up based on the animation call order and durations.

The Glow framework builds a set of helper methods for common animation tasks from a core animation class called glow.anim.Animation. This can be combined with glow.anim.Timeline to create complex animations.

Queues and Events

Animation frameworks usually use `setInterval` and `clearInterval` interval to sequence events. This can be combined with custom events. Due to JavaScript's single-threaded environment, `setInterval` and events are the best way of managing the asynchronous nature of animations.

Animation Basics

As we've seen, animation frameworks build on top of CSS property manipulation. At its most basic, animation looks like this:

```
// Box CSS: #box { background-color: red; width: 20px; height: 20px; position: absolute; left: 0; top: 10 }

function animate() {
  var box = document.getElementById('box'),
      duration = 1000,
      start = (new Date).valueOf(),
      finish = start + duration,
      interval;

  interval = setInterval(function() {
    var time = (new Date).valueOf(), frame = time > finish ? 1 : (time - start) / duration;

    // The thing being animated
    box.style.left = frame * 100 + 'px';

    if (time > finish) {
      clearInterval(interval);
    }
  }, 10);
}
```

I based this code on emile.js. It uses `setInterval` which calls a function every few milliseconds. Time calculations (based on the number of milliseconds returned by `new Date`) are used to set up the animation and stop it. Once the end of the animation has been reached, `clearInterval` is called.

This is essentially what jQuery's animation library does with its queues.

Time-Based Animation

Take a look at this JavaScript animation example:

```
function animate() {
  var box = document.getElementById('box'),
      duration = 1000,
      start = (new Date).valueOf(),
      finish = start + duration,
      interval;

  interval = setInterval(function() {
    var time = (new Date).valueOf(), position = time > finish ? 1 : (time - start) / duration;
    box.style.left = position * 100 + 'px';
    if (time > finish) {
      clearInterval(interval);
    }
  }, 10);
}
```

This makes a div move across the screen. This animation runs every 10 milliseconds, and uses the current time to determine the animation's progress.

The reason this technique is often used for animations in JavaScript is timers compete for attention with other parts of the browser. JavaScript runs in a single thread, but execution is shared by other timers and events. Time is sliced up depending on what needs attention.

Technically `setInterval` could be called a set number of times, based on the desired animation duration. In reality the overall duration would change depending on what else is going on in the browser.

Using `setInterval` with `Date` isn't such a big problem though. Each call to `valueOf` is fairly minimal.

To understand more about how timers work in JavaScript, read through How JavaScript Timers Work by John Resig.

Animating Properties

Manipulating CSS properties is the core part of animation frameworks. Once this is done, Turing can make animation helpers available so common tasks are succinct.

The `animate()` example above uses the `left` style property to move the item by multiplying the current animation `position` by a value. The `position` value is between 0 and 1. This value is actually the end point of the animation. If we wanted to move something by saying "increase margin-left to 50", we could multiply `position` by 50 for each frame.

That means we need to be able to pass `animate` style properties and values. We typically think in terms of stylesheet values rather than DOM properties, so some frameworks parse CSS properties into valid DOM properties.

Values themselves need to be translated. For example, `50px` or `2em` need to have the units extracted so the numbers can be manipulated, then added back again:

```
{ 'marginLeft' : '10px' }
// Becomes:
parsedStyles['marginLeft'] = { number: '10', units: 'px' }
// Then the transform looks like this:
for (var property in parsedStyles) {
  element.style[property] = parsedStyles[property].number * position + parsedStyles[property].unit
}
```

Simply allowing properties to be collected together in objects allows multiple properties to be animated at once.

Parsing Style Values

To get the values into the format above, I've simply used `parseFloat` and `String.prototype.replace`:

```
function parseCSSValue(value) {
  var n = parseFloat(value);
  return { number: n, units: value.replace(n, '') };
}
```

This can't cope with many types of values — colours won't work for example, but it will work for animating positions.

API

The Turing animation API currently looks like this:

```
turing.anim.animate(element, 1000, { 'marginLeft': '8em', 'marginTop': '100px' });
```

Durations are in milliseconds and are a required parameter. There’s also a fourth parameter that isn’t shown here which will allow callbacks to be specified.

I haven’t bothered converting CSS property names into DOM names because it seemed like unnecessary complexity.

The full code is available in GitHub in turing.anim.js.

Easing

Easing is an important animation technique that most people will never realise exists. It’s a surprisingly important technique for creating natural-looking animations, even for simple web animations. Most animations use easing functions, and animation frameworks usually use one by default. Animations can seem strangely abrupt without non-linear easing.

The script.aculo.us wiki has a useful interactive example of many common easing functions. Another great resource is in the documentation for Tweener.

From wikipedia:

> “Ease-in” and “ease-out” in digital animation typically refer to a mechanism for defining the ‘physics’ of the transition between two animation states, eg. the linearity of a tween.

Adding Easing Function Support

Currently the `turing.anim.animate` method accepts parameters but doesn’t do anything with them. Let’s change it to work like this:

```
turing.anim.animate(box, 1000, { 'marginLeft': '8em', 'marginTop': '100px' }, { easing: 'bounce'
```

The default easing will be linear, and I’ll include a few easing functions so other people can reference them and create their own:

```
var easing = {};
easing.linear = function(position) {
  return position;
};
```

Then the `animate` method just needs to ensure the user has specified either a string or a function:

```
if (options.hasOwnProperty('easing')) {
  if (typeof options.easing === 'string') {
    easingFunction = easing[options.easing];
  } else {
    easingFunction = options.easing;
  }
}
```

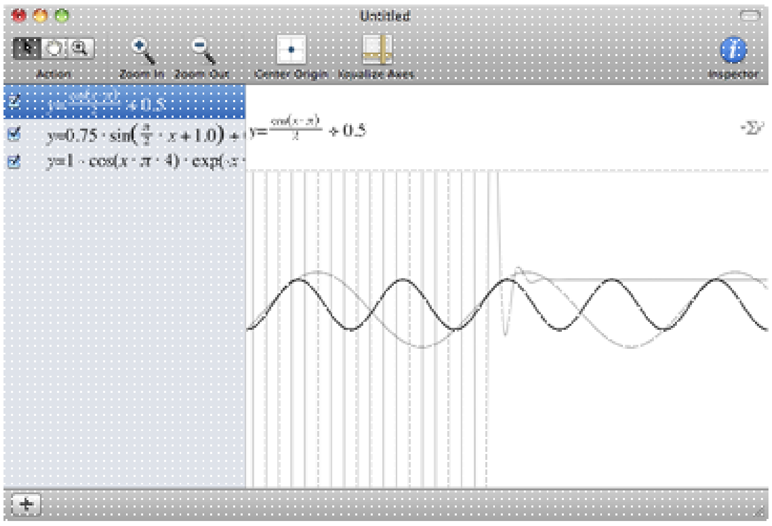
Writing Easing Functions

The classic easing function is based on `cos`:

```
easing .sine  = function  (position  ) {
  return   (-Math .cos (position   *  Math .PI )  /  2)  +  0.5 ;
};
```

Mathematical functions can be exploited to create interesting effects. Functions like `sin` and `cos` are oscillators, and their periodic nature is useful for many types of animation. The visual impact of this easing function is more like acceleration than linear movement.

If you'd like to create your own functions based on trigonometry, it's worth reading the wikipedia page on sine waves to get a feel for the basics first.


Untitled
Action
Zoom In
Zoom Out
Center Origin
Equalize Axes
Inspector
$y=0.75\cdot\sin(\frac{\pi}{2}\cdot x+1.0)+$
$y=1\cdot\cos(x\cdot\pi\cdot 4)\cdot\exp($
$y=\frac{\cos(x\cdot\pi)}{2}+0.5$

I explored the easing functions I've used here with Grapher, which comes with some versions of Mac OS. If you're familiar with the mathematical notation for basic trigonometric equations it can be a useful tool for finding out why a particular function makes everything move backwards suddenly.

More "programmatic" equations can be created using `if` statements to change the position multiplier at certain points:

```
easing .bounce   = function  (position  ) {
  if  (position   <  (1  /  2.75 ))  {
    return   7.6  *  position   *  position   ;
  } else  if  (position   <  (2  /2.75 ))  {
    return   7.6  *  (position   -=  (1.5  /  2.75 ))  *  position   +  0.74 ;
  } else  if  (position   <  (2.5  /  2.75 ))  {
    return   7.6  *  (position   -=  (2.25  /  2.75 ))  *  position   +  0.91 ;
  } else  {
    return   7.6  *  (position   -=  (2.625  /  2.75 ))  *  position   +  0.98 ;
  }
};
```

I based these equations on the Tweener library and script.aculo.us.

Animation Helpers

Our API currently looks like this:

```
turing .anim .animate (element , 1000 , { 'marginLeft' : '8em' , 'marginTop' : '100px' }, { easing : 'bounc
```

This is quite unnatural though. It's weird to have to think in terms of CSS properties, and most web sites and apps just need a handful of common effects:

- Fade — fade an element by changing its opacity
- Highlight — rapidly change an element's background colour to draw attention to it
- Movement — move an element

Fade

Ideally we want to be able to specify a core fade function to build fade in and out:

```
turing .anim .fade (element , duration , { 'from' : '8em' , 'to' : '100px' , 'easing' : easing });
```

Easing should be optional.

Using the existing animation API, the core of the `fade` function should look like this:

```
element .style .opacity = options .from ;
anim .animate (element , duration , { 'opacity' : options .to }, { 'easing' : options .easing })
```

Now `fadeIn` and `fadeOut` can be built with sensible defaults:

```
anim .fadeIn = function (element , duration , options ) {
  options = options || {};
  options .from = options .from || 0.0 ;
  options .to = options .to || 1.0 ;
  return anim .fade (element , duration , options );
};
```

`fadeOut` needs to use its own easing function to change the `from` and `to` values negatively:

```
anim .fadeOut = function (element , duration , options ) {
  var from ;
  options = options || {};
  options .from = options .from || 1.0 ;
  options .to = options .to || 0.0 ;

  // Swap from and to
  from = options .from ;
  options .from = options .to ;
  options .to = from ;

  // This easing function reverses the position value and adds from
  options .easing = function (p) { return (1.0 - p) + options .from ; };

  return anim .fade (element , duration , options , { 'easing' : options .easing });
};
```

This might seem a bit confusing. Recall the `animate` function's time value scaling equation:

```
element .style [property ] = (easingFunction (position ) * properties [property ]. number ) + properties [p
```

This scales values using elapsed time to the number passed in the DOM style properties. In this case it's the value for from.

Great! But this won't work in IE, and Turing doesn't have any browser detection features.

Opacity and IE

IE uses filters to set opacity, like this: `filter: alpha(opacity=0-100)`. Let's slightly change the way styles are set to use a function that is aware of browser differences:

```
// opacityType = (typeof document.body.style.opacity !== 'undefined') ? 'opacity' : 'filter'

function  setCSSProperty  (element , property , value ) {
  if (property  == 'opacity'  && opacityType  == 'filter' ) {
    element .style [opacityType ] = 'alpha(opacity='   + Math .round (value  * 100 ) + ')' ;
    return  element ;
  }
  element .style [property ] = value ;
  return  element ;
}
```

Another slight glitch in IE's opacity handling is elements need *layout*. I've changed the part of the code that parses CSS values to set `zoom` to fix this:

```
for (var property  in properties ) {
  if (properties .hasOwnProperty  (property )) {
    properties [property ] = parseCSSValue  (properties [property ]);
    if (property  == 'opacity'  && opacityType  == 'filter' ) {
      element .style .zoom  = 1;
    }
  }
}
```

You can read more about layout in On having layout by Ingo Chao.

Colour Support

I mentioned that I wanted to build in a helper for highlighting an element (the Yellow Fade Technique). The problem with this is the library doesn't currently support colour parsing.

The reason colour parsing is required is we need to convert values to `rgb(0, 0, 0)` notation: most people will expect to be able to pass hexadecimal colours.

I based our colour parser on Stoyan Stefanov's parser here: RGB color parser in JavaScript. The coding style is particularly suitable for this tutorial series, because it presents the concepts in easy to read code.

The core of the colour parser is an array of regular expressions and functions that can extract numbers for a given format and turn it into RGB values:

```
Colour .matchers  = [
  {
    re : /^rgb\((\d{1,3}),\s*(\d{1,3}),\s*(\d{1,3})\)$/       ,
    example : ['rgb(123, 234, 45)'  , 'rgb(255,234,245)'  ],
    process : function  (bits ){
      return  [
        parseInt (bits [1]) ,
        parseInt (bits [2]) ,
        parseInt (bits [3])
      ];
    }
  },
```

Notice the `example` property documents how the matcher works which is a great touch that I lifted directly from Stoyan's code (which I think he got from Simon Willison's date library).

Now the parser just needs to loop through each matcher. There are only 3, and colour parsing will only be used outside of the main animation loop:

```
Colour.prototype.parse = function() {
  var channels = [], i;
  // Loop through each matcher
  for (i = 0; i < Colour.matchers.length; i++) {
    channels = this.value.match(Colour.matchers[i].re);
    // An array of numbers will be returned if the value was matched by the regex
    // else null
    if (channels) {
      // Set each value to r, g, b properties in this object
      channels = Colour.matchers[i].process(channels);
      this.r = channels[0];
      this.g = channels[1];
      this.b = channels[2];
      break;
    }
  }
  this.validate();
}
```

I could have used an array for each property here, but I thought I'd be explicit and use `r, g, b` properties for each value. This means there is some code repetition later, but at least you can easily see what's going on.

Something else I took from Stoyan's library is the validation code:

```
Colour.prototype.validate = function() {
  this.r = (this.r < 0 || isNaN(this.r)) ? 0 : ((this.r > 255) ? 255 : this.r);
  this.g = (this.g < 0 || isNaN(this.g)) ? 0 : ((this.g > 255) ? 255 : this.g);
  this.b = (this.b < 0 || isNaN(this.b)) ? 0 : ((this.b > 255) ? 255 : this.b);
}
```

This just makes sure values are between 0 and 255.

Channel Surfing

Something I realised early on when developing this was channels should be able to move independently. Imagine if you wanted to animate this:

```
// From:
rgb(55, 255, 0);

// To:
rgb(255, 0, 10);
```

The red, green and blue properties are moving in different directions. Our easing-based transforms so far just multiply values by position, so we need something that can:

- Figure out the "direction" of each colour
- Correctly multiply each colour by the current position in the animation

Direction can be represented by -1 and 1. We don't need to conditionally check for direction though, the values can just be multiplied by the direction:

```
nextRed = startRed + (redDirection * (Math.abs(startRed - endRed) * easingFunction(position)))
```

`Math.abs` is used here to make sure the value isn't negative. Technically the start value might be less than the end value — we just need to know the magnitude:

```
155 - 55
// 100
55 - 155
// -100
Math.abs(55 - 155)
100
```

Transformations

We might find CSS values are numbers or colours. To handle this, I've made Turing normalise properties we want to animate into a format that includes a "transformation" function. These functions know how to correctly manipulate colours or numbers. This was already hinted at because we needed the basis to write out units (px, em, etc.)

The core animation loop now includes a reference to a `transform` property that the CSS value parser sets.

The colour transform looks like this:

```
function colourTransform(v, position, easingFunction) {
  var colours = [];
  colours[0] = Math.round(v.base.r + (v.direction[0] * (Math.abs(v.base.r - v.value.r) * easingFunc
  colours[1] = Math.round(v.base.g + (v.direction[1] * (Math.abs(v.base.g - v.value.g) * easingFunc
  colours[2] = Math.round(v.base.b + (v.direction[2] * (Math.abs(v.base.b - v.value.b) * easingFunc
  return 'rgb(' + colours.join(', ') + ')';
}
```

It doesn't bother using `new Colour(...)` to set up a new colour because we just care about the RGB values here.

I got the idea for normalised CSS values into objects that contain unit and value presentation functions from Emile.

Highlight Helper

The highlight helper function can now be written:

```
anim.highlight = function(element, duration, options) {
  var style = element.currentStyle ? element.currentStyle : getComputedStyle(element, null);
  options = options || {};
  options.from = options.from || '#ff9';
  options.to = options.to || style.backgroundColor;
  options.easing = options.easing || easing.sine;
  duration = duration || 500;
  element.style.backgroundColor = options.from;
  return setTimeout(function() {
    anim.animate(element, duration, { 'backgroundColor': options.to, 'easing': options.easing })
  }, 200);
};

// Usage:

turing.anim.highlight(element);
```

It sets a bunch of defaults then sets a bright yellow background colour. After that it animates fading the background colour to the original colour. `getComputedStyle` has to be used to get the current colour.

There might be instances where passing `options.from` is a better idea.

Conclusion

Animating CSS values is difficult because we need to detect what type of value we're dealing with, parse it, change it, then correctly update the DOM with the new values. By keeping the parsed results non-specific the animation loop can be kept decoupled from the code that does the real work.

Eventually the CSS-related code should be moved to `turing.css`, because it could be useful outside of animations.

To play with the code and make some experiments of your own, the easiest way is to check out the source, cd into `test` and open up `anim_test.html` in a browser and `anim_test.js` in an editor.

Movement Helper

We need to be able to do this:

```
turing .anim .move (element , duration , { x: '100px' , y: '100px'  });
```

And the element should move 100 pixels across and down from its current position. The API we've built so far will do this for relative positioned elements with the following code:

```
anim .animate (element , duration , { 'left' : options .x, 'top' : options .y });
```

Using an absolute positioned element will cause it to jump to the first point in the animation. That means we need to move absolute elements *relative* to their original location. I'll come back to this later.

Chained API

Ideally animation calls, whether they be from helpers or `animate`, should work when chained. Chaining calls should create a sequence of animations:

```
turing .anim .chain (element )
  .highlight ()
  .move (1000 , { x: '100px' , y: '100px'  })
  .animate (2000 , { height : '0px'  });
```

It would be useful to be able to pause animations, too:

```
turing .anim .chain (element )
  .highlight ()
  .pause (2000 )
  .move (1000 , { x: '100px' , y: '100px'  })
  .animate (2000 , { width : '1000px'  })
  .fadeOut (2000 )
  .pause (2000 )
  .fadeIn (2000 )
  .animate (2000 , { width : '20px'  })
```

I've used as similar technique to the Enumerable library's Chainer class (part 5). This isn't reused here but it could be made more generic.

The basic principle is to wrap each chainable method in a function that can correctly sequence the animation calls. The `anim` object contains all of the relevant methods; this object can be iterated over:

```
for  (methodName   in  anim ) {
  (function  (methodName  ) {
    var  method  = anim [methodName  ];
    // ...
  })( methodName  );
}
```

The anonymous function causes the `methodName` variable to get bound to the scope. The `Chainer` class needs to keep track of the element we're animating, and the current position in time:

```
Chainer   = function  (element  ) {
  this .element   = element  ;
  this .position   = 0;
};
```

Then as the animation progresses, we need to update the position:

```
for  (methodName   in  anim ) {
  (function  (methodName  ) {
    var  method  = anim [methodName  ];
    Chainer  .prototype  [methodName  ] = function  ()  {
      this .position   +=  current   position  ;
    };
  })( methodName  );
}
```

`setTimeout` can be used to schedule the animations:

```
setTimeout  (function  ()  {
  method .apply (null ,  args );
}, this .position  );
```

This will not block, so all of the chained animations will be started at around the same time. Since `position` is incremented with each animation's duration, the animations should fire at approximately the right time.

The only other thing left to do is progress the `arguments` passed to the function to insert the element.

Putting it Together

```
Chainer   = function  (element  ) {
  this .element   = element  ;
  this .position   = 0;
};

for  (methodName   in  anim ) {
  (function  (methodName  ) {
    var  method  = anim [methodName  ];
    Chainer  .prototype  [methodName  ] = function  ()  {
      var  args  = Array .prototype  .slice .call (arguments  );
      args .unshift  (this .element  );
      this .position   +=  args [1]  ||  0;
      setTimeout  (function  ()  {
        method .apply (null ,  args );
      }, this .position  );
      return  this ;
    };
  })( methodName  );
}

anim .chain  =  function  (element  ) {
  return  new  Chainer  (element  );
};
```

CSS3

CSS3 introduces several modules for dealing with animation. All of the relevant specifications are currently working drafts, with a lot of input from Apple. Therefore, if you're interested in trying these examples WebKit-based browsers (and particularly Safari in Mac OS) are currently the best way of trying them out.

The specifications are:

- CSS Transitions Module Level 3
- CSS 2D Transforms Module Level 3
- CSS 3D Transforms Module Level 3
- CSS Animations Module Level 3

CSS3 Transitions

Transitions are useful because CSS properties can be animated, much like our framework code. The syntax is fairly easy to follow:

```
.example  {
-webkit -transition  : all  1s  ease -in-out ;
-moz -transition  : all  1s  ease -in-out ;
-o-transition  : all  1s  ease -in-out ;
-webkit -transition  : all  1s  ease -in-out ;
transition  : all  1s  ease -in-out ;
}
```

The `all` instruction refers to the CSS property to animate — you could single out `margin` if required.

```
<style>
#move  {
  background-color   : #ff0000  ;
  width : 100px ;
  height : 100px ;
-webkit -transition  : all  5s  ease -in-out ;
-moz -transition  : all  5s  ease -in-out ;
-o-transition  : all  5s  ease -in-out ;
transition  : all  5s  ease -in-out ;
}

#move-container   :hover  #move  {
  margin-left  : 440px ;
  width : 200px
}
</style>

<div  id= "move-container"   >
  <div  id= "move" >
  </div>
</div>
```

The previous example should create a red box that stretches and moves (a *translation*) over 5 seconds.

CSS3 Transforms

There are several different transforms:

```
transform  : translate  (50px ,  200px );
transform  : rotate  (45deg );
transform  : skew (45deg );
transform  : scale (1,  2);
```

These transforms can be combined with `transition` to animate for an event, like hovering over an element.

CSS3 Animations

Animations aren't well supported right now, but they do provide a simple way of creating *keyframes* for the animation over time. The CSS looks like this:

```
@keyframes   'wobble'   {

  0% {
    left :  100 px ;
  }

  40 %  {
    left :  150 px ;
  }

  60 %  {
    left :  75 px ;
  }

  100 %  {
    left :  100 px ;
  }

}
```

I think it's likely that many JavaScript animation frameworks will leave out support for animations, since they'll have equivalent functionality already.

Animation Performance Problems

In QtWebkit and Graphics at webkit.org, the authors discuss the problems caused when trying to animate using the box model. Animating margin, padding, background, outline, and border will all result in relatively slow animations.

The CSS Animations Module Level 3 is seen as a way around traditional animation performance issues. The working draft's editors are all from Apple, and people are still arguing about whether or not animation belongs in CSS or JavaScript. However, it's currently the only way of getting hardware acceleration that I'm aware of.

Hardware Acceleration

Most computers and devices feature specialised graphics chips. If hardware acceleration sounds mysterious to you, it's nothing more than using these chips instead of the CPU. The browser has to perform a huge amount of work to calculate changes when animating elements, but offloading this work to software and hardware that deals with graphics reduces CPU load.

CSS3 animations should look noticeably slicker on devices with slower CPUs like the iPhone.

Feature Detection

We generally test browser support by checking if an object or property is available, rather than forking code based on the browser's version strings. There are no properties that can give away hardware acceleration support, so in Scripty2 Thomas Fuchs does this:

```
function isHWAcceleratedSafari () {
  var ua = navigator.userAgent, av = navigator.appVersion;
  return (!ua.include('Chrome') && av.include('10_6')) ||
   Prototype.Browser.MobileSafari;
}
```

CSS3 support is detected by checking if `WebKitTransitionEvent` or `MozTransition` exists:

```
var div = document.createElement('div');

try {
  document.createEvent("WebKitTransitionEvent");
  supported = true;

  hardwareAccelerationSupported = isHWAcceleratedSafari();
} catch (e) {
  if (typeof div.style.MozTransition !== 'undefined') {
    supported = true;
  }
}

div = null;
```

Then there's the problem of translating CSS3 animation properties to vendor-specific properties.

The state of CSS3 support is currently in flux because the standards aren't ready yet. Most browsers still use vendor prefixed tags, which means we need to know what browser we're dealing with.

Detecting browser support for CSS3 is a little bit tricky, but it's not impossible. WebKit browsers have an event object called `WebKitTransitionEvent`, and Opera uses `OTransitionEvent`. Firefox has a style attribute called `MozTransition`.

I've created an object with a list of properties that can be used to query vendor support:

```
// CSS3 vendor detection
vendors = {
  // Opera Presto 2.3
  'opera': {
    'prefix': '-o-',
    'detector': function() {
      try {
        document.createEvent('OTransitionEvent');
        return true;
      } catch (e) {
        return false;
      }
    }
  },

  // Chrome 5, Safari 4
  'webkit': {
    'prefix': '-webkit-',
    'detector': function() {
      try {
        document  .createEvent   ('WebKitTransitionEvent'     );
        return   true ;
      } catch (e)  {
        return   false ;
      }
    }
  },

  // Firefox 4
  'firefox'  : {
    'prefix'  :  '-moz-'  ,
    'detector'  :  function  ()  {
      var  div  =  document  .createElement   ('div' ),
          supported   =  false ;
      if  (typeof   div .style .MozTransition    !==  'undefined'  )  {
        supported   =  true ;
      }
      div  =  null ;
      return   supported  ;
    }
  }
};

function   findCSS3VendorPrefix    ()  {
  for  (var  detector   in  vendors  )  {
    detector   =  vendors  [detector  ];
    if  (detector  ['detector'  ]())  {
      return   detector  ['prefix'  ];
    }
  }
}
```

Move Animation Implementation

To use CSS3 for move animations, we need to do the following:

- Detect when a CSS property is being used to move an element
- Get the vendor prefix
- Set up CSS3 style properties instead of using Turing's animation engine

Detecting when a CSS Property Means Move

The convention I've been using is to manipulate the `left` or `top` style properties to move an element. Whenever these properties are animated and a vendor prefix has been found, then we can use CSS transitions.

The best place to do this is in the property loop inside `anim.animate`:

```
for  (var  property   in  properties   )  {
  if  (properties   .hasOwnProperty    (property  ))  {
    properties   [property  ] =  parseCSSValue    (properties   [property  ],  element   ,  property  );
    if  (property    ==  'opacity'    &&  opacityType    ==  'filter'  )  {
      element   .style  .zoom   =  1;
    }  else  if  (CSSTransitions    .vendorPrefix    &&  property    ==  'left'   ||  property    ==  'top' )  {
      // Do CSS3 stuff here and return before Turing animates with its own routines
    }
  }
}
```

I've stolen `camelize` from Prototype to make writing out CSS easier:

```
element  .style [camelize  (this .vendorPrefix    +  'transition'   )]  =  property    +  ' '  +  duration    +  'ms '   +  (
element  .style [property  ]  =  value ;
```

In the case of Firefox 4, this would translate to:

```
element  .style [MozTransition   ]  =  'left 1000ms linear'    ;
element  .style ['left' ]  =  '100px'  ;
```

I've put this in a function called `start`, and I've also added an `end` function to clear the transition afterwards:

```
CSSTransitions    = {
  // ...

  start : function  (element  ,  duration  ,  property  ,  value ,  easing )  {
    element  .style [camelize  (this .vendorPrefix    +  'transition'   )]  =  property   +  ' '  +  duration   +  'ms '
    element  .style [property  ]  =  value ;
  },

  end : function  (element  ,  property  )  {
    element  .style [camelize  (this .vendorPrefix    +  'transition'   )]  =  null ;
  }
};
```

The core of the property loop now looks like this:

```
CSSTransitions   .start (element  ,  duration  ,  property  ,  properties  [property  ]. value  +  properties  [property
setTimeout  (function  ()  {  CSSTransitions   .end (element  ,  property  );  },  duration  );
return ;
```

References

- emile.js
- Getting Started with Emile
- Scriptaculous Wiki
- Scriptaculous effects.js
- Glow's anim.js
- Glow's animtimeline example
- MooTools' Fx
- jQuery's effects.js
- jQuery Animation
- How Timers Work
- Wikipedia on Inbetweening:
- script.aculo.us wiki on transitions
- Tweener's interactive easing library
- Tweener
- On having layout
- RGB color parser in JavaScript
- Effect.Highlight in scriptaculous

Touch

Supporting Touchscreen Devices

Libraries like jQTouch help support devices like the iPhone, iPad and Android phones. A lot of people are interested in this because building web apps is arguably easier than writing native Objective-C or Java code, and in some cases a web app might be more suitable. In fact, I've noticed a lot of companies building jQTouch interfaces as a way of exploring an iPhone version of their site or app before actually building a native app.

jQTouch is a jQuery plugin that provides a whole wealth of features, and also includes graphical components to make building native-looking apps easier. In this tutorial series, I'm going to focus on supporting the more low-level features, like touchscreen events.

A Simple Example: Orientation

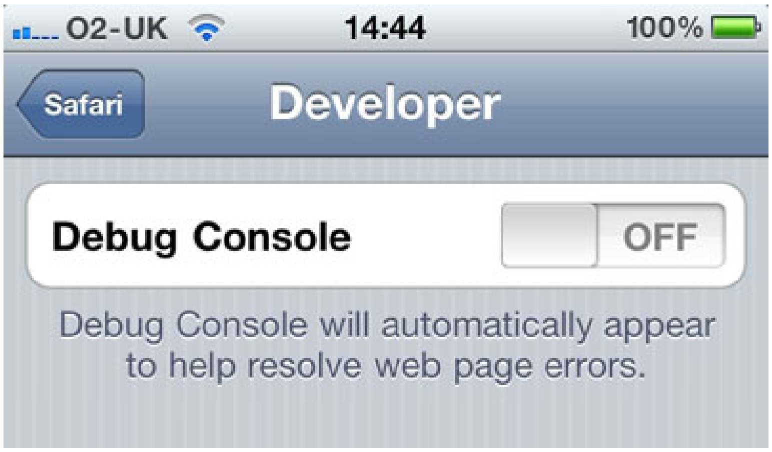
To kick things off, let's look at detecting orientation changes. DailyJS's Turing framework already has support for events. WebKit supports the `orientationchange` event:

```
turing .events .add ($t ('body' )[0], 'orientationchange'   , function  (e) {
  alert ('put me down you oaf!'    );
});
```

Debugging

Before progressing, let's look at debugging options. Android devices can use `logcat` — take a look at Android Debug Bridge for some help in this area.

Safari for iPhone has a debug console. To enable it, go to Settings, Safari, Developer and enable debugging:



Debug Console OFF

Debug Console will automatically appear to help resolve web page errors.

Then each page gets a debug bar:

Orientation Property

There’s a property called `orientation` on `window` that can be used to detect the current angle of the device. This can be interpreted to figure out the exact orientation of the device:

```
touch .orientation    = function  ()  {
  var  orientation    = window .orientation   ,
       orientationString     =  '';
  switch   (orientation   )  {
    case  0:
      orientationString     +=  'portrait'  ;
    break ;

    case  -90:
      orientationString     +=  'landscape right'    ;
    break ;

    case  90:
      orientationString     +=  'landscape left'    ;
    break ;

    case  180:
      orientationString     +=  'portrait upside-down'     ;
    break ;
  }
  return   [orientation   ,  orientationString     ];
};
```

This code is from the `turing.touch` object. Now orientation changes can be detected and handled like this:

```
turing  .events  .add ($t ('body' )[0],  'orientationchange'    ,  function  (e)  {
  alert (turing  .touch  .orientation   ());
});
```

Remember that `$t` is the shortcut for `turing.dom.get` — this is all in `turing.alias.js`.

Events

We’ve seen how to detect orientation changes. This is actually very simple once you know how to interpret `window.orientation`. Other events, like multi-touch gestures, take a bit more work. jQTouch, which is one of the leading frameworks in this area, makes this easier by offering helper events like `swipe` and `tap`. The `swipe` event also makes it easy to detect the direction of the swipe.

The jQTouch source also has a joke about touch events:

```
// Private touch functions (TODO: insert dirty joke)
function touchmove (e) {
```

The “real” events, as far as Safari is concerned, are:

- `touchstart`
- `touchmove`
- `touchend`
- `touchcancel`

The callback methods are passed event objects with these properties:

- `event.touches`: All touches on the page
- `event.targetTouches`: Touches for the target element
- `event.changedTouches`: Changed touches for this event

The `changedTouches` property can be used to handle multi-touch events.

State

The way I handle tap and swipe events is by recording the state at each event.

- If there’s just one event and the position hasn’t changed, it’s a tap event
- If `touchmove` fired, work out the distance of the movement and how long it took for a swipe

I’m fairly sure that only horizontal swipes make sense, seeing as vertical movement scrolls the browser window.

From the developer’s perspective, the API should look like this:

```
turing.events.add(element, 'tap', function(e) {
  alert('tap');
});
turing.events.add(element, 'swipe', function(e) {
  alert('swipe');
});
```

We can watch for all the touch events inside the library, then fire tap or swipe on the event’s target element. The library registers for events like this:

```
turing.events.add(document, 'touchstart', touchStart);
turing.events.add(document, 'touchmove', touchMove);
turing.events.add(document, 'touchend', touchEnd);
```

The `touchStart` and similar methods are our own internal handlers. That’s where tap and swipe events are detected. I’ve actually put these “global” handlers in a method called `turing.touch.register` because I don’t yet have a good way of adding them unless they’re needed.

I thought it might be nice if `turing.events.add` could allow other libraries to extend it, so the touch library could say “hey, if anyone wants events called tap or touch, run register first.”

State and Pythagoras

When `touchStart` is fired, I store the state of the event:

```
function touchStart (e) {
  state .touches   = e.touches ;
  state .startTime    =  (new  Date ). getTime  ();
  state .x  =  e.changedTouches   [0]. clientX  ;
  state .y  =  e.changedTouches   [0]. clientY  ;
  state .startX   =  state .x;
  state .startY   =  state .y;
  state .target   =  e.target  ;
  state .duration   =  0;
}
```

Quite a lot of things are recorded here. I got the idea of working out the duration of events from jQTouch — it makes sense to do things based on time when working with gestures.

Single taps are a simple case:

```
function   touchEnd  (e)  {
  var  x  =  e.changedTouches   [0]. clientX  ,
       y  =  e.changedTouches   [0]. clientY  ;

  if  (state .x  ===  x  &&  state .y  ===  y  &&  state .touches  .length   ==  1)  {
    turing  .events  .fire (e.target  ,  'tap' );
  }
}
```

Moves are a bit more complicated. I use Pythagoras to calculate how far the finger has moved. This probably isn’t really required, but I like bringing highschool maths into my tutorials if possible:

```
function   touchMove  (e)  {
  var  moved   =  0,  touch   =  e.changedTouches   [0];
  state .duration   =  (new  Date ). getTime  ()  -  state .startTime  ;
  state .x  =  state .startX   -  touch  .pageX ;
  state .y  =  state .startY   -  touch  .pageY ;
  moved   =  Math .sqrt (Math .pow (Math .abs (state .x),  2)  +  Math .pow (Math .abs (state .y),  2));

  if  (state .duration   <  1000  &&  moved   >  turing  .touch .swipeThreshold   )  {
    turing  .events  .fire (e.target  ,  'swipe'  );
  }
}
```

I calculate `turing.touch.swipeThreshold` based on screen resolution. I was thinking about scaling up the minimum distance considered a swipe to the iPhone 4’s high resolution, but then I found out that it treats the browser as if it was the old iPhone resolution, so this wasn’t actually required.

The `state` object isn’t global, it’s wrapped up inside a good old closure, like the rest of the class. You can check it all out in turing.touch.js

Chained APIs

Introduction

What we want to be able to do is chain finder methods:

```
turing ('.example'  ). find ('p' )
```

This would find things with the class example, then the associated paragraphs. We can use `turing.dom.get` to implement the core functionality, but `get()` does not accept a "root" element, so we'll need to add that.

Another thing is, calling `turing()` makes no sense, because it isn't a function. Let's address that while we're at it.

The alias module will also have to be changed, because it currently wraps `turing.dom.get` anyway.

Tests

The implementation should satisfy the following test in `test/dom_test.js`:

```
given ('chained DOM calls'    ,  function  ()  {
  should  ('find  a  nested tag'    ,  turing  ('.example3'  ). find ('p' ). length ). equals  (1);
});
```

I should cover more methods and cases, but I'm on a tight schedule here!

Updating Core

This is simpler that you might expect. The core module currently exposes `turing` as an object with a bunch of metadata properties. This can be changed to a function to get the jQuery-style API. The only issue is I don't want to make `turing.dom` a core requirement.

To get around that I'm going to allow an init method to be overridden from outside core. This could be handled in a better way to allow other libraries to extend the core functionality, but let's do it like this for now:

```
function   turing  ()  {
  return   turing  .init .apply  (turing  ,  arguments  );
}

turing  .VERSION   =  '0.0.28'  ;
turing  .lesson   =  'Part 28: Chaining'    ;
turing  .alias  =  '$t' ;

// This can be overriden by libraries that extend turing(...)
turing  .init   =  function  ()  {  };
```

Then in the DOM library:

```
turing  .init   =  function  (selector  )  {
  return   new  turing  .domChain  .init (selector  );
};
```

This last snippet is based on the fakeQuery example.

Updating turing.dom

This is all completely taken from the fakeQuery example. The real `find` method in `turing.domChain` (which came from `fakeQuery.fn`) looks like this:

```
find : function (selector ) {
  var elements = [],
      ret = turing (),
      root = document ;

  if (this .prevObject ) {
    if (this .prevObject .elements .length > 0) {
      root = this .prevObject .elements [0];
    } else {
      root = null ;
    }
  }

  elements = dom .get (selector , root );
  this .elements = elements ;
  ret .elements = elements ;
  ret .selector = selector ;
  ret .length = elements .length ;
  ret .prevObject = this ;
  ret .writeElements ();
  return ret ;
}
```

It depends on `dom.get` for the real work, which I covered way back in part 6 (and onwards).

The `writeElements` method sets each element to a numerical property, so the `Array`-like API is available:

```
$t ('.example3' ). find ('p' )[ 0 ]
```

I also added a shorthand `first()` method to the same class while I was at it.

DOM Root

Setting a “root” element for `dom.get` looks like this:

```
dom .get = function (selector ) {
  var tokens = dom .tokenize (selector ). tokens ,
      root = typeof arguments [1] === 'undefined' ? document : arguments [1],
      searcher = new Searcher (root , tokens );
  return searcher .parse ();
};
```

An undefined property will become `document`, which means it can accept `null`. I had to make the existing `find` methods check for `null` as a special case.

Namespaces and Chaining

Throughout this series I’ve referenced techniques used by widely-used frameworks like jQuery and Prototype. Prototype packs a lot of functionality and extends global JavaScript objects to do this.

jQuery takes a different approach. It uses large module-like chunks of functionality wrapped in closures, then specific parts are exposed through the `jQuery` object (we usually write `$()` instead).

Turing has been designed in a similar way to jQuery — to carefully keep implementation details private and make functionality available without polluting global objects.

One drawback of our current implementation is everything takes a lot of typing. Disregarding the alias we created, code looks like this:

```
var element = turing.dom.get('#events-test a')[0];
turing.events.add(element, 'click', callback);

// Or...
turing.events.add(turing.dom.get('#events-test a')[0], 'click', callback);
```

We'd do this in jQuery:

```
$('#events-test').click(function() {
  // Handler
});
```

In this case, `click` is a shortcut, so the following is equivalent:

```
$('#events-test').bind('click', (function() {
  // Handler
});
```

This chaining can go on as long as you want. jQuery even provides tools for popping up to different points in a chained result stack, like `end()`:

```
$('ul.first').find('.selected')
  .css('background-color', 'red')
.end().find('.hover')
  .css('background-color', 'green')
.end();
```

This works particularly well when working with DOM traversal.

What this style of API gives us is the safety of namespaced code with the power and succinctness of prototype hacking, without actually modifying objects that don't belong to us.

API

The way this works in jQuery is `jQuery()` accepts a selector and returns an array-like **jQuery object**. The returned object has a `length` property, and each element can be accessed with square brackets. It's not a true JavaScript `Array`, just something similar enough.

Each call in the chain is operating on a `jQuery` object, which means all of the appropriate methods are available.

Previously…

We've already seen a combination of aliasing and currying to create a chainable API in Turing — check out *turing.enumerable.js* and *turing.anim.js*. In these cases, API calls were chained based on the first parameter — the first parameter for functions in these classes was always a certain type, so we could shortcut this and create a chain.

This is really a case of currying, and is one of those fine examples of a nice bit of functional programming in JavaScript.

fakeQuery

jQuery's chaining is based around the DOM, so the previous examples don't really help. Rather than jumping straight into Turing code, I've created a little class you can play with called `fakeQuery`. This will illustrate what underpins jQuery.

It uses a mock up of the DOM so it has something to query:

```
var  dom  =  [
  {  tag :  'p' ,  innerHTML  :  'Test 1'  ,  color  :  'green'   },
  {  tag :  'p' ,  innerHTML  :  'Test 2'  ,  color  :  'black'   },
  {  tag :  'p' ,  innerHTML  :  'Test 3'  ,  color  :  'red'  },
  {  tag :  'div' ,  innerHTML  :  'Name: Bob'    }
];
```

It's not a particularly accurate representation of the DOM, but it's readable.

This is the core function:

```
function   fakeQuery  (selector  )  {
  return   new  fakeQuery  .fn .init (selector  );
}
```

It returns a new object based on an `init` method. The `init` method builds an object which can carry around the current selector and related elements:

```
fakeQuery  .fn  =  fakeQuery  .prototype   =  {
  init :  function  (selector  )  {
    this .selector   =  selector  ;
    this .length   =  0;
    this .prevObject   =  null ;

    if  (!selector  )  {
      return   this ;
    }  else  {
      return   this .find (selector  );
    }
  },

  find :  function  (selector  )  {
    // Finds elements
    // Returns a new fakeQuery
  },

  color :  function  (value )  {
    // Creates a copy of the current elements
    // Changes them
    // Returns a fakeQuery object with these elements
  }
};

fakeQuery  .fn .init .prototype   =  fakeQuery  .fn ;
```

The `prevObject` property could be used to implement `end()` (mentioned above). The full code is in a gist: fakeQuery. This code uses Node, but you could delete the Node-related parts if you want to run it with Rhino.

Running this code with something like `fakeQuery('p').color('red').elements` will produce:

```
[  {  tag :  'p' ,  innerHTML  :  'Test 1'  ,  color  :  'red'  }
,  {  tag :  'p' ,  innerHTML  :  'Test 2'  ,  color  :  'red'  }
, { tag : 'p' , innerHTML  : 'Test 3'  , color : 'red'  }
]
```

Overall Pattern

The overall architecture of jQuery is deceptively simple:

- A “container function” is used to create new objects without having to type `new`
- It returns objects based on a CSS selector
- A class is created and copied so usage of methods like `find` can be called in a chain

The key to the last part is `fakeQuery.fn.init.prototype = fakeQuery.fn;`. This line is what allows the `init` method reference `fakeQuery.prototype`. You can try running the code without this if you want to see what happens.

Chained Events

API Design

We want to be able to do this:

```
turing ('#element'  ). bind ('click'  , function  (e) {
  alert ('Stop clicking me!'    );
});
```

If you haven’t read the other chaining tutorials, this might not seem interesting. The reason we’re doing this is to get a chainable API for DOM finders, like jQuery. So multiple finders could be called:

```
turing ('#element'  ). find ('.element li a'   ). bind ('click'  , function  (e) {
  alert ('Stop clicking me!'    );
});
```

Adding events with Turing is performed with `turing.events.add(element, 'event name', callback)`. I’ll use the method name `bind` instead of `add` so it doesn’t look confusing next to DOM-manipulation code.

Test

We need this test to pass (in test/events_test.js):

```
should ('bind events using the chained API'      , function  ()  {
  var  clicks   = 0;
  turing  ('#events-test a'   ). bind ('click'  , function  ()  { clicks ++;  });
  turing  .events  .fire (element  , 'click' );
  return   clicks ;
}). equals  (1);
```

Running it right now results in an error:

> **should bind events using the chained API**: 1 does not equal: TypeError: Result of expression ‘turing(’#events-test a’).bind’ [undefined] is not a function.

Implementation

It seems like we can just alias `bind` to `add` with a bit of currying, but that doesn’t fit with our style of keeping each module independent (else `turing.dom` will rely on `turing.events`).

However, previously we exposed the object that is returned through the chained DOM calls: `turing.domChain`. Let’s try extending that from the events API if it’s available.

In turing.events.js:

```
events .addDOMethods    = function  ()  {
  // If there's no domChain then the DOM module hasn't been included
  if  (typeof   turing .domChain   ===  'undefined'  )  return ;

  // Else it's safe to add the bind method
  turing .domChain  .bind  =  function  (type ,  handler )  {
    var  element   = this .first ();
    if  (element )  {
      turing .events  .add (element  ,  type ,  handler );

      // NOTE: "this" refers to the current domChain object,
      //       which contains the stack of elements
      return  this ;
    }
  };
};

// It's safe to always run addDOMethods when
// the events module is loaded
events .addDOMethods   ();
```

I’ve commented each part, but it’s fairly straightforward.

Event Handler Shortcuts and Loop Scoping

The only DOM event handler I added to our chained API was `click`. Let’s add more of the events named after the HTML ‘on’ attributes. I used the list on MDC’s Event Handlers page as a reference, and set up aliases like this:

```
events .addDOMethods    = function  ()  {
  if  (typeof   turing .domChain   ===  'undefined'  )  return ;

  turing .domChain  .bind  =  function  (type ,  handler )  {
    var  element   = this .first ();
    if  (element )  {
      turing .events  .add (element  ,  type ,  handler );
      return  this ;
    }
  };

  var  chainedAliases    =  ('click dblclick mouseover mouseout mousemove '          +
                           'mousedown mouseup blur focus change keydown '          +
                           'keypress keyup resize scroll'      ). split (' ' );

  for  (var  i  =  0;  i  <  chainedAliases   .length  ;  i++)  {
    (function  (name )  {
      turing .domChain  [name ]  =  function  (handler )  {
        return  this .bind (name ,  handler );
      };
    })( chainedAliases   [i]);
  }
};
```

The technique I used to create an array from a string is found throughout jQuery. It's handy because it has less syntax than lots of commas and quotes. I use the anonymous function to capture the name parameter for each alias, doing it with `var name = chainedAliases[i]` would bind `name` to the last value executed, which isn't what we want.

In jQuery's code they use `jQuery.each` to iterate over the event names, which actually reads better. I put our iterators in turing.enumerable.js and have been avoiding interdependence between modules, so I'm doing it the old fashioned way.

However, doing it this way does illustrate an interesting point about JavaScript's lexical scoping and closures. Try this example in a prompt or browser console:

```
var methods = {},
    items = ['a', 'b', 'c'];

for (var i = 0; i < items.length; i++) {
  var item = items[i];

  methods[item] = function() {
    console.log('Item is: ' + item);
  };
}

console.log('After the for loop, item is: ' + item);

for (var name in methods) {
  console.log('Calling: ' + name);
  methods[name]();
}
```

This will result in:

```
After the for loop, item is: c
Calling: a
Item is: c
Calling: b
Item is: c
Calling: c
Item is: c
```

Why is each item set to 'c' when each function is called? In JavaScript, variables declared in `for` are in the same scope rather than a new local scope. That means there aren't three `item` variables, there is just one. And this is why jQuery's version is more readable. I've edited this version of jQuery's events.js to illustrate the point:

```
jQuery.each(aliases, function(i, name) {
  jQuery.fn[name] = function(data, fn) {
    return this.bind(name, data, fn);
  };
});
```

Variables declared inside `jQuery.each` are effectively in a different scope, and of course the `name` parameter passed in on each iteration is the one we want.

Trigger vs. Bind

Calling `jQuery().click()` without a handler actually fires the event, which I've always liked. Can we do something similar?

We just need to check if there's a handler in `turing.domChain.bind`:

```
if (handler) {
  turing.events.add(element, type, handler);
} else {
  turing.events.fire(element, type);
}
```

While I was looking at `turing.domChain.bind` I changed it to bind to all elements instead of the first one. I thought that way felt more natural.

You could do a quick test with this:

```
$t('p').click(function(event) {
  event.target.style.backgroundColor = '#ff0000'
});
```

It'll bind to all of the paragraphs instead of just the first one.

Feature Detection

querySelectorAll

The selector engine we built for the core of `turing.dom` was based on the way Firefox interprets CSS selectors. I liked the approach for the context of these tutorials, because it's a very pragmatic approach that's easy to follow.

Browsers have been shipping with querySelectorAll for a while, which reduces the amount of work required to implement DOM lookups.

That means `turing.dom.get` could be rewritten:

```
// Original code minus all the
// magic in the Searcher class and tokenizer
dom.get = function(selector, root) {
  var tokens = dom.tokenize(selector).tokens,
      searcher = new Searcher(root, tokens);
  return searcher.parse();
};

// New selector API
dom.get = function(selector) {
  return document.querySelectorAll(selector);
};
```

But not all browsers support this yet, so let's check if it's available:

```
dom.get = function(selector) {
  var root = typeof arguments[1] === 'undefined' ? document : arguments[1];
  if ('querySelectorAll' in document) {
    return root.querySelectorAll(selector);
  } else {
    return get(selector, root);
  }
};
```

In the Wild

jQuery will check for `querySelectorAll`, but it'll only use it under certain conditions. This is from jQuery 1.4.2:

```
if (document.querySelectorAll) {
  (function(){
    var oldSizzle = Sizzle, div = document.createElement("div");
    div.innerHTML = "<p class='TEST'></p>";

    // Safari can't handle uppercase or unicode characters when
    // in quirks mode.
    if ( div.querySelectorAll && div.querySelectorAll(".TEST").length === 0 ) {
      return;
    }

    Sizzle = function(query, context, extra, seed){
      context = context || document;

      // Only use querySelectorAll on non-XML documents
      // (ID selectors don't work in non-HTML documents)
      if ( !seed && context.nodeType === 9 && !isXML(context) ) {
        try  {
          return  makeArray  (  context  .querySelectorAll   (query ),  extra  );
        }  catch (e){}
      }

      return  oldSizzle  (query ,  context  ,  extra ,  seed );
    };

    for  (  var  prop  in  oldSizzle   )  {
      Sizzle [  prop  ]  =  oldSizzle  [  prop  ];
    }

    div  =  null ;  // release memory in IE
  })();
}
```

The use of an element for capability detection is common in frameworks — sometimes it's the only reliable way of detecting a browser's behaviour.

It's a little bit different in Dojo 1.5, but the same Safari issue is mentioned:

```
 // some versions of Safari provided QSA, but it was buggy and crash-prone.
  // We need te detect the right "internal" webkit version to make this work.
  var  wk  =  "WebKit/"  ;
  var  is525  =  (
    d.isWebKit   &&
    (nua .indexOf  (wk )  >  0)  &&
    (parseFloat  (nua .split  (wk )[1])  >  528 )
  );

  // IE QSA queries may incorrectly include comment nodes, so we throw the
  // zipping function into "remove" comments mode instead of the normal "skip
  // it" which every other QSA-clued browser enjoys
  var  noZip  =  d.isIE  ?  "commentStrip"    :  "nozip"  ;

  var  qsa  =  "querySelectorAll"    ;
  var  qsaAvail   =  (
    !!getDoc  ()[ qsa ]  &&
    // see #5832
    (!d.isSafari   ||  (d.isSafari   >  3.1 )  ||  is525   )
  );

  //Don't bother with n+3 type of matches, IE complains if we modify those.
  var  infixSpaceRe    =  /n\+\d|([^ ])?([>~+])([^ =])?/g      ;
  var  infixSpaceFunc    =  function  (match ,  pre ,  ch ,  post )  {
    return  ch  ?  (pre  ?  pre  +  " "  :  "" )  +  ch  +  (post  ?  " "  +  post  :  "" )  :  /*n+3*/  match ;
  };

  var  getQueryFunc    =  function  (query ,  forceDOM  ){
    //Normalize query. The CSS3 selectors spec allows for omitting spaces around
    //infix operators, >, ~ and +
    //Do the work here since detection for spaces is used as a simple "not use QSA"
    //test below.
    query  =  query .replace  (infixSpaceRe   ,  infixSpaceFunc   );

    if(qsaAvail  ){
      // if we've got a cached variant and we think we can do it, run it!
      var  qsaCached   =  _queryFuncCacheQSA    [query ];
      if(qsaCached   &&  !forceDOM  ){  return  qsaCached  ;  }
    }

    // Snip
```

Here the browser and version are derived from the user agent string.

In the Wild

A good example of a feature detection library is has.js.

> Browser sniffing and feature inference are flawed techniques for detecting browser support in client side JavaScript. The goal of has.js is to provide a collection of self-contained tests and unified framework around using pure feature detection for whatever library consumes it.

Using has.js as inspiration, we should be able to rewrite the previous code like this:

```
dom.get = function(selector) {
  var root = typeof arguments[1] === 'undefined' ? document : arguments[1];
  return turing.detect('querySelectorAll') ?
    root.querySelectorAll(selector) : get(selector, root);
};
```

Feature Detection Implementation

Making a library of feature tests is fairly easy with a plain `Object`. Tests can be referred to by name and easily looked up at runtime. Also, a cache can be used to store the results of the tests.

The core of this functionality is something inherent to JavaScript programming rather than just browser-related, so I put this in turing.core.js:

```
var testCache = {},
    detectionTests = {};

turing.addDetectionTest = function(name, fn) {
  if (!detectionTests[name])
    detectionTests[name] = fn;
};

turing.detect = function(testName) {
  if (typeof testCache[testCache] === 'undefined') {
    testCache[testName] = detectionTests[testName]();
  }
  return testCache[testName];
};
```

The results are cached because they should only be run once. This type of capability detection is intended to be used against the environment, rather than features that might load dynamically in runtime, so I think it’s safe to run the tests once.

Then the jQuery-inspired `querySelectorAll` test can be added in turing.dom.js:

```
turing.addDetectionTest('querySelectorAll', function() {
  var div = document.createElement('div');
  div.innerHTML = '<p class="TEST"></p>';

  // Some versions of Safari can't handle uppercase in quirks mode
  if (div.querySelectorAll) {
    if (div.querySelectorAll('.TEST').length === 0) return false;
    return true;
  }

  // Helps IE release memory associated with the div
  div = null;
  return  false ;
});
```

Conclusion

Looking through popular JavaScript frameworks made me realise that most of them still use the user agent string to determine what capabilities are available. This might be fine most of the time, but I don’t consider it best practice. The way jQuery tests capabilities using techniques like the dummy DOM element creation seems like a lot of work, but it relies on browser behaviour rather than the user agent string.